The Complete Guide to

C#

Programming

Authorised for Sale in
Indian Sub-Continent Only

dreamtech Press

Er. V.K. Jain

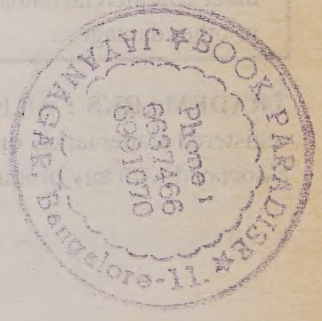
The Complete Guide to

C#

Programming

Er. V.K. Jain
BE (E), M. Tech, FIE, FIETE
Chartered Engineer

Programs Compiled by
Hemlata



Published by

dreamtech Press

19-A, Ansari Road, Daryaganj,
New Delhi-110002

©Copyright 2001 by Dreamtech Press, 19-A, Ansari Road, Daryaganj, New Delhi-110002

This book may not be duplicated in any way without the express written consent of the publisher, except in the form of brief excerpts or quotations for the purposes of review. The information contained herein is for the personal use of the reader and may not be incorporated in any commercial programs, other books, databases, or any kind of software without written consent of the publisher. Making copies of this book or any portion for any purpose other than your own is a violation of copyright laws.

LIMITS OF LIABILITY/DISCLAIMER OF WARRANTY : The author and publisher have used their best efforts in preparing this book. The author make no representation or warranties with respect to the accuracy or completeness of the contents of this book, and specifically disclaim any implied warranties of merchantability or fitness of any particular purpose. There are no warranties which extend beyond the descriptions contained in this paragraph. No warranty may be created or extended by sales representatives or written sales materials. The accuracy and completeness of the information provided herein and the opinions stated herein are not guaranteed or warranted to produce any particulars results, and the advice and strategies contained herein may not be suitable for every individual. Neither Dreamtech Press nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

TRADEMARKS : All brand names and product names used in this book are trademarks, registered trademarks, or trade names of their respective holders. Dreamtech Press is not associated with any product or vendor mentioned in this book.

First Edition : 2001

First Reprint Edition : 2001

Printed at : Unique Color Carton., Mayapuri, New Delhi-64.

Preface

C#(C Sharp) is a modern, object-oriented language that combines the simplicity of Visual Basic with the power and flexibility of C++. C# has been specifically designed by Microsoft to be the language of choice for writing applications for the new .NET platform—the first development platform designed exclusively for the www (world wide web) or internet.

In writing this book, I have kept in mind the great future and the immense potential of the C# language. Since the book is intended for those who desire to gain a thorough understanding of C# Programming, I have taken special care to incorporate the subtle nuances of this prospering language.

This book will be of great help to programmers who are already familiar with programming in C, C++ or VB. They can effortlessly upgrade their skills through this and achieve greater heights in the world of computer programming.

The book begins with a chapter containing an introduction and an over-all-view of C#. The ensuing three chapters discuss the .Net Framework and the other platforms with regard to C#. From the fourth chapter onwards, the book covers every feature of the C# language in detail. Apart from C#, I have undertaken special efforts to cover Microsoft's .NET Framework & Platform, XML, ASP.NET and interfacing with COM in detail. Further, the book also details on Exceptions and Error Handling, Data Handling and various projects in C#. In spite of being a highly technical subject, I have made every effort to make the writing lucid and interesting. Sufficient codings also accompany the subject to help the reader understand the practical aspects of the language.

I hope that the reader stands to benefit from going through this book. Herein I would like to take the opportunity and wish every earnest reader, a bright and prosperous career.

I would like to offer my sincere thanks to the various Microsoft web sites which proved to be valuable sources of information. Finally, this work owes immensely to the efforts of the Technical Editors of Dreamtech Press and the Programmers of Dreamtech Software for their meticulous work in shaping up this book.

—Author

Contents

1 Introduction and Overview of C# 6

1.1 NET Programming Language 1
1.2 Evolution of C# 1
1.3 Features of C# 2
1.4 Benefits of Changing Over to C# 3
1.4.1 C# is a part of .NET Platform 3
1.4.2 Direct Components for WEB-Services 3
1.4.3 RAPID Development Tool 4
1.4.4 Emphasis on Productivity and Safety 4
1.4.5 Eliminates Costly Programming Errors 4
1.4.6 Better Mapping between Business Process and Implementation 4
1.4.7 XML Based Metadata 5
1.4.8 Extensive Interoperability 5

2 Microsoft's .NET Framework & Platform

2.1. Distributed Computing 7
2.2 Three Accelerators of Distributed Computing 7
2.3 Overview of the .NET Platform 8
2.4 Uses of .Net Microsoft 9
2.5 .NET and its Components 9
2.5.1 Microsoft's .NET Strategy 10
2.5.2 The .NET Platform Tools 10
2.6 .NET Framework and Visual Studio.NET 10
2.7 Server Infrastructure 10
2.8 Smart Devices 11
2.9 .NET Experiences 11
2.9.1 What is an Example of a .NET Experience? 12
2.10 Microsoft .NET Benefits 13
2.11 How Microsoft .NET Changes Computing 14
2.11.1 Changes for End Users 14
2.11.2 Changes for Enterprises 15
2.11.3 What Stays The Same 15
2.12 Next Generation of Computing 16
2.13 Visual Studio.NET 16
2.14 The .NET Framework 16
2.15 Visual Studio .NET Highlights 17
2.16 Getting Started 19
2.17 Common Language Runtime (CLR) 19
2.17.1 Services Provided by CLR 20
2.17.2 In-process and Cross-process Communication Work in the
2.18 Common Type System (CTS) 22
2.19 Common Language Specification (CLS) 22
2.20 Microsoft Intermediate Language (MSIL) 22
2.21 Managed Code and Managed Data 22
2.22 Assemblies 23
2.22.1 What are Private Assemblies and Shared Assemblies 23
2.22.2 Difference between a Namespace and an Assembly Name 24
2.23 Interoperability of COM Objects and .NET 25
2.24 How scalable is .NET? 25
2.25 .NET : Security and Privacy Concerns 25
2.26 .NET Vs XML Platforms 26
2.27 Advantages of .NET Programming Model and Tools 26
2.28 Application Deployment and Isolation 27

2.29 Application Domain 27
2.30 Windows in .NET 27
2.31 Digital Nervous System 28
2.32 .NET Framework SDK 28
2.33 Microsoft .NET Framework SDK Tools & Utilities 28
2.33.1 Configuration and Deployment Tools & Utilities 29
2.33.2 Windows Forms Design Tools & Utilities 29
2.33.3 Security Tools & Utilities 30
2.33.4 General Tools & Utilities 30
2.34 Tools Required 31
2.35 VB.Net 32
2.36 SERVLET ENGINES JRUN, ASP AND JSP 33
2.37 NGWS 34

3 C# AND OTHER PLATFORMS

3.1 Object Oriented Language Design 37
3.2 Procedural Languages 37
3.3 Object-Oriented Programming 37
3.4 Encapsulation 39
3.5 Information Hiding 39
3.6 Abstraction and Encapsulation 40
3.7 Inheritance 40
3.8 Polymorphism 40
3.9 Advantages of OOPs 41
3.10 C++ and other Object Oriented programming (OOPs) Languages 41
3.11 Evolution of C++ Programming Language 41
3.12 Distinctive Features of C++ Language 42
3.13 Limitations of Using c++ 42
3.14 Basic programming Features of C++ Language 43
3.14.1 Commments 43
3.14.2 Input & output 44
3.14.3 Variables 44
3.14.4 LOOPs 44
3.14.5 Constants 44
3.14.6 Conditional statements 44
3.14.7 Relational Operators 45
3.14.8 Repeating 46
3.15 Java 46
3.16 Comparison of C#, C++ and VB.net 47
3.17 XML and Java 49

4 C# : INTRODUCTION

4.1 Introduction to C Sharp 51
4.2 Main: Entry Point 52
4.3 First C# Program 54
4.4 The Main Method 55
4.5 Another C# Program 55
4.6 Program Output 56
4.7 Printing and Formatting Output 59
4.8 Input from Console 60
4.9 Compilation and Execution 61
4.10 Namespace Declaration 61
4.11 Getting Command-Line Input 63
4.12 Interactive Input / Output 64
4.13 Command Line Outputs 65
4.14 Main with return type void and int 66
4.15 Common Language Platform : (CLR) 67

4.16 Two ways of getting output 67
4.17 Output Formatting Options 67
4.18 Making Project in Visual C#.Net 68
4.19 General Structure of a C# Program 72
4.19.1 Program Startup 73
4.19.2 Program Termination 73
4.19.3 Declarations 74
4.20 Arguments in Main method 74
4.21 Runtime Time Arguments 76
4.22 command-line arguments Samples 78

5 TYPES in C#

5.1 Types 79
5.2 Value Types 80
5.3 Default Constructors 81
5.4 Struct Types 81
5.4.1 Simple Types 81
5.4.2 Integral Types 82
5.4.3 Floating Point Types 83
5.4.4 The Decimal Type 84
5.4.5 The Bool Type 87
5.5 Enumeration Types 87
5.6 Reference Types 87
5.7 Class Types 88
5.8 The Object Type 88
5.9 The String Type 89
5.10 Interface Types 89
5.11 Array Types 89
5.12 Delegate Types 89
5.13 Predefined Types 89
5.14 The concept of Boxing and Unboxing 93
5.15 Conversions 94
5.16 Array Types 96
5.17 Type System Unification 96
5.18 Variables and Parameters 97
5.19 Automatic Memory Management 102
5.20 Operands 104
5.21 Statements 105
5.21.1 Statement lists and block statements 105
5.21.2 Labeled statements and goto statements 105
5.21.3 Local constant declarations 105
5.21.4 Local variable declarations 105
5.21.5 Expression statements 106
5.21.6 if statements 106
5.21.7 switch statements 106
5.21.8 while statements 106
5.21.9 do statements 107
5.21.10 for statements 107
5.21.11 foreach statements 107
5.21.12 break statements 107
5.21.13 continue statements 107
5.21.14 return statements 108
5.21.15 throw statements and try statements 108
5.21.16 checked and unchecked statements 108
5.21.17 lock statements 108

5.21.18 using statements 108
5.22 Variables 109
5.22.1 Variable Categories 109
5.22.2 Static Variables 110
5.22.3 Instance Variables 110
5.22.4 Array Elements 110
5.22.5 Value Parameters 110
5.22.6 Reference Parameters 111
5.22.7 Output Parameters 111
5.22.8 Local Variables 111
5.23 Expressions 112
5.23.1 Expression Classifications 112
5.24 Operators 112
5.24.1 Unary Operators 113
5.24.2 Operator Precedence and Associativity 115
5.24.3 Operator Overloading 116
5.25 Boxing And Unboxing 120
5.25.1 Unboxing Conversions 120
5.26 Type Declarations 122
5.26.1 Types Description 122
5.27 Value Data Types 123

6 CLASSES

6.1 C# Classes 127
6.2 Objects 128
6.2.1 Objects Defined 129
6.2.2 Member Data and Member Functions 129
6.2.3 Objects and Instances 129
6.3 Classes 131
6.4 Objects as Data types 132
6.5 Creating Classes 132
6.5.1 The Class Declaration 132
6.5.2 The Class Body 133
6.5.3 Classes and Members 133
6.5.4 Scope of Class Members: Public, Protected, and Private 134
6.5.5 Using a Class (Instantiation) 135
6.5.6 Instantiating an Object 135
6.6 Using an object's member functions 135
6.7 Most simple C# Program with a class 136
6.8 Static Methods and new Operator 137
6.9 Creating an Object 138
6.9.1 Static Methods 140
6.10 Providing Constructors for Your Classes 142
6.11 Class Declaration 146
6.12 All Types of Class Members 148
6.12.1 Constants 150
6.12.2 Fields 151
6.12.3 Methods 152
6.12.4 Properties 153
6.12.5 Events 154
6.12.6 Operators 155
6.12.7 Indexers 157
6.12.8 Instance constructors 158
6.12.9 Destructors 159
6.12.10 Static constructors 159
6.13 Inheritance 160

6.14 Controlling Access to Members of a Class 161
6.14.1 Private 162
6.14.2 Protected 163
6.14.3 Public 164
6.14.4 Package 165
6.15 Cleaning Up Unused Objects 166
6.16 The Garbage Collector 166
6.17 Finalization 166
6.18 Implementing Classes 166
6.19 Class Library and Name Space 169

7 Methods

7.1 Methods 171
7.2 Structure of a Method 176
7.3 Method Overloading 182
7.4 Implementing Methods 182
7.5 Class Containing Functions 182
7.6 A Simple Calculator Class 185
7.13 Try and Catch 189

8 STATEMENTS AND CONTROL

8.1 Controls and Statements 195
8.2 Statements 196
8.3 Selection statements 196
8.4 The if statement 196
8.5 The Syntax of the if Statement 199
8.6 If-else statement 201
8.7 Nested ifs 201
8.8 if-then-else 204
8.9 The Switch Statement 209
8.9.1 Syntax of switch Statement 211
8.10 Loop control 214
8.10.1 Types of Loops 214
8.10.2 Counters. 214
8.10.3 Loop Control Variable 214
8.11 Iteration Statements 215
8.12 The While Statement 215
8.12.1 Syntax of While Statement 218
8.13 The Do Statement 219
8.13.1 The Syntax of do-statement: 221
8.14 The For Statement 222
8.14.1 Syntax of for Statement 226
8.15 The foreach Statement 227
8.15.1 Syntax of foreach-statement: 228
8.16 Jump Statements 229
8.17 The Break Statement 230
8.18 The Continue Statement 231
8.19 The Goto Statement 231
8.20 The Return Statement 234
8.20.1 Syntax of return-statement: 236
8.21 Common Programming Errors related with control structuring 236
8.22 Summary of C control statements 237

9 Abstract and Base Classes

9.1 Abstract Modifier 239
9.2 Abstract classes 241

9.3 Virtual 243
9.4 Virtual Methods 245
9.5 Internal 253
9.6 Base Classes 254
9.6.1 Namespaces for Base Classes 255
9.7 Introduction to the .NET Base Classes 272
9.7.1 Collection Classes 273
9.8 foreach Statement 277
9.8.1 Using foreach with Arrays 278
9.8.2 Using foreach with Collections 279

10 STRUCTS AND ENUM

10.1 Structs 285
10.2 Structs Types 288
10.3 Struct Declarations 289
10.4 Struct Modifiers 289
10.5 Struct Interfaces 289
10.6 Enums 290
10.7 Enumerator Base Types 291
10.7.1 Initialization 291
10.7.2 Enums Declaration 292
10.8 Enum Modifiers 293
10.9 Enum Members 293
10.10 Enum Values and Operations 296
10.11 Browsing the Internet 300
10.12 Accessing the Registry 302

11 ARRAYS

11.1 Arrays 305
11.2 Concept of Arrays 305
11.3 Single-dimensional Arrays 307
11.4 Passing Arrays as Parameters 307
11.5 Array Initialization 310
11.5.1 Passing Arrays as Parameters 312
11.6 Array Types Based on Dimensions 314
11.7 Declaring Arrays 316
11.8 Creating Arrays 317
11.9 Accessing Array Members 318
11.10 Arrays are Objects 318
11.11 Using foreach on Arrays 319
11.12 Definite Assignment and Arrays 320
11.13 Array Types 321
11.14 The System.Array Type 321
11.15 Array Members 321
11.16 Array Covariance 322
11.17 Array Initializers Revisited 322
11.18 ArrayList 324
11.18.1 Adding data to an ArrayList 324
11.18.2 Delete data from an ArrayList 324
11.18.3 Searching elements in ArrayList 324
11.18.4 Difference in handling Arraytypes 324
11.19 out and ref 328
11.19.1 Collection classes 333
11.20 Foreach Statement 336
11.20.1 Using foreach with Collections 337
11.20.2 Features of the foreach Statement 342

12 STRINGS

12.1 Strings 345
12.2 String Operations 345
12.3 Converting Objects to Strings 346
12.4 StringBuilder 347
12.5 Regular Expressions 348
12.6 File and Folder Operations 350
12.7 Finding Out Information About a File 351
12.8 Listing Files in a Folder 352
12.9 Copying and Deleting files 353
12.10 Reading Text Files 354
12.11 Writing Text Files 355
12.12 Reading Binary Files 356
12.13 Writing Binary Files 357
12.13.1 Creating A File 358
12.13.2 Strings 359
12.13.3 String literals and operations in C# 360
12.13.4 When is Clone not Clone? 362
12.13.5 String equality/comparison methods 363
12.13.6 Intern/IsInterned 365
12.14 Picture formats 368
12.14.1 How Formatting Works 370
12.14.2 Formatted Strings in .NET 376
12.15 STRING FORMATTING IN .NET 376
12.16 User Defined Numeric Formats 378
12.17 Floating Point Formats 379
12.17.1 WriteLine 379
12.18 Predefined Formats 379
12.19 Polymorphic Formatting 380
12.20 Design/Implementation Approach 382

13 ATTRIBUTES

13.1 Attributes 389
13.2 Creating Custom Attributes 391
13.3 Retrieving Attribute Information 392
13.4 Using Attributes 394
13.5 Attribute Usage 397
13.6 Attribute Pickling 397
13.7 An Attribute of Your Own 397
13.8 Attribute Parameters 397
13.9 Attribute Parameter Types 398
13.10 Reflecting on Attributes 398
13.11 Interfaces 401
13.12 Delegates 403
13.13 Delegate Types 404
13.14 Delegate Declarations 409
13.15 Combinable Delegate Types 410
13.16 Delegate Instantiation 410
13.17 Multi-Cast Delegates 410
13.18 Delegate Invocation 410
13.20 Interfaces 411
13.21 Delegates vs. Interfaces 413
13.22 Name Spaces 413
13.23 Types of Namespaces 423
13.23.1 Compilation Units 423
13.24 Namespace Declarations 424

13.25 Using Directives 425
13.26 Using Namespace Directives 426
13.27 Namespaces and Assemblies 426
13.28 Versioning 428
13.29 Uses of Versioning 431
13.30 A Component in C# 432
13.31 Preprocessor Directives 440
13.32 Preprocessor Definition 440

14 XML

14.1 XML Technology Backgrounder 443
14.2 Uses of XML 443
14.3 Advantages of XML 444
14.4 Sample XML Data 444
14.5 Elements and Attributes 445
14.6 XML Schemas 445
14.7 Style Sheets 445
14.8 XML Elements 445
14.9 XML Attributes 446
14.10 Introduction to XML Schemas and XSD 446
14.11 Advantages of XSD 447
14.11.1 Sample XSD 447
14.11.2 XSD Schemas and DataSets 448
14.12 XML Examples 448
14.13 Compilation 450
14.14 Explanation 450
14.15 Output 451
14.16 Processing the XML File 452
14.17 Tags for Documentation Comments 454

15 Building Projects

15.1 Projects in Visual Basic and C# 455
15.2 Project Items 456
15.3 Adding Project Items 456
15.4 Local Project Items 457
15.5 Web Project Items 457
15.6 Web Projects 458
15.6.1 Web Project File Storage 458
15.6.2 Web Project Build Model 458
15.6.3 Server Requirements for Web Projects 458
15.6.4 Web Access Methods 458
15.6.5 Choosing the Best Access Method 459
15.6.6 When to Specify the Web Access Method 459
15.6.7 Adding Projects to Source Control 460
15.7 Project Behavioral Model 460
15.7.1 Project Behavioral Models, Classified and Differentiated 460
15.7.2 Project Type Behavioral Model 461
15.7.3 Moving and Copying Items 461
15.8 Saving Project Items 462
15.9 Removing Project Items 463
15.10 Referencing 463
15.11 Working with Projects 464
15.12 Renaming a Project Web Reference 465
15.13 Updating a Project Web Reference 465
15.14 Adding Web Reference 466
15.15 Creating New Solutions and Projects 467
15.16 Solutions and Projects 467

15.17 Projects and Solutions 467
15.18 Solution Directory Structure 468
15.19 Solution Explorer 468
15.20 Solution Explorer Toolbar Icons 469
15.21 Signal Icons in Solution Explorer 470
15.22 Multi-Project Solutions 470
15.23 Solution Items 471
15.24 Miscellaneous Files 471
15.25 Current Selection Context 472
15.26 Solutions, User Interface Elements 472
15.27 Add Existing Project Dialog Box 473
15.28 Preview 475
15.29 Arrange Icons 475
15.30 Introduction 475
15.31 Declaring the Constructor 476
15.32 About the Samples 478
15.33 Making a clock 480

16 EXCEPTIONS AND ERROR HANDLING IN C#

16.1 Exception Handling in C# 483
16.2 The throw Statement 484
16.3 How exceptions are handled 485
16.4 The try Statement 485
16.5 The System.Exception class 488
16.6 Common Exception Classes 488
16.7 The Checked and Unchecked Statements 488
16.8 The Lock Statement 489
16.9 The Using Statement 490
16.10 Trying and Catching 491
16.11 Hierachical Structure of Exception 492
16.12 Passing Exceptions to Caller 494
16.13 User defined Exception Clause 497
16.14 Error Handling Using Exceptions 498
16.15 Ways of Reporting and Handling Errors Return Codes 499
16.16 Exceptions 499
16.17 What is the relevance of exceptions in this context? 500
16.18 Using try, catch, and throw 500
16.19 What if you need to do some cleanup? 502
16.20 When to Use Which 504
16.21 Inside the .NET Runtime: How Exceptions Work 504
16.22 Types of Exception Handlers 504
16.23 The Exception Object 505
16.24 What Exception Should I Throw? 505

17 DATA HANDLING IN C#

17.1 ADO 507
17.2 Introduction to D ata Access with ADO.NET 511
17.3 ADO.NET Uses a Disconnected Data Architecture 516
17.4 Data is Cached in Datasets 517
17.5 Datasets are Independent of Data Sources 518
17.6 Data is Persisted as XML 518
17.6.1 Advantages of using XML 519
17.7 Schemas Define Data Structures 519
17.8 Components of ADO.NET 520
17.9 Programming Architecture 520

17.10 DataSet Namespace 520
17.11 Typed versus Untyped Datasets 521
17.12 Related Tables and Relation Objects 521
17.13 Updating Datasets and Data Stores 522
17.14 Connection Objects 523
17.15 Table Mappings 524
17.16 Reading and Updating with DataSet Commands 524
17.17 Concurrency Control 526
17.18 ADOCommand and SQLCommand 526
17.19 ADODataReader and SQLDataReader 526
17.20 Databases 526
17.21 Objectives behind maintaining databases: 527
17.22 Relational database 527
17.23 SQL (Structured Query Language) 527
17.24 Microsoft's SQL Server 528
17.25 Accessing Data with ADO.NET 528
17.26 Building .NET Applications 528
17.27 Connecting to Database 529
17.28 Accessing an Access Database using ADO 530
17.29 ADO+ : The Evolution of the Data Species 531
17.30 Creating a Distributed Application 532
17.31 Process for Creating a Distributed Application 532
17.32 Create the Middle-Tier Business Object 533
17.33 Create and ConFig. a DataSet 534
17.34 Create the User Interface 535
17.35 Windows User Interface Option 536
17.36 Web User Interface Option 537
17.37 Deploying the Solution 538
17.38 A Common Data Manipulation Language 539
17.39 The Data Species 540
17.40 ADO Plus 541
17.41 ADO+ Constituent Elements 543
17.42 ADO+ Commands 544
17.43 Strongly Typed Programming Language 545
17.44 Speciality of ADO+ 546
17.45 How to Import ADO 546
17.46 Working with Recordsets in .NET 548
17.47 Server Cursors 552
17.48 The ADO+ Object Model 552
17.49 Managed Providers 552
17.50 The DataSet 554
17.51 Populating DataSets with DataSetCommands 556
17.52 XML Integration 556
17.53 The XmlDataDocument 557
17.54 Visual Studio .NET Data Designers 558
17.55 Typed DataSets 559
17.56 Connecting to a text file using ADO.NET 561

18 ASP.NET

18.1 Active Server Pages 565
18.2 Features of ASP 565
18.3 Active Server Pages and Security 566
18.4 Compatibility of ASP with other Applications 566
18.5 ASP .NET - A New Operating System 567
18.6 ASP.NET and Compatibility 568
18.7 Introduction to ASP.NET 569
18.8 ASP+ Platform 569

18.9 ASP+ Page Framework 570
18.10 ASP+ Web Services 570
18.11 ASP+ Deployment 571
18.12 ASP.NET Features 571
18.13 Web Services 572
18.14 User Controls (Formerly Pagelets) 572
18.15 Registration of Components 573
18.16 Mobile Controls 573
18.17 Configuration Details 574
18.18. Getting Started with ASP.NET 575
18.19 Requirement for running ASP.NET 577
18.20 How is ASP.NET different from ASP 3.0? 577
18.21 Create Dynamic Content Using ASP 577
18.22 Working with Properties and Data Types in ASP.NET 578
18.23 Facilites Provided by ASP.NET 579
18.24 Introduction to Web Forms 580
18.25 Web Services Overview 583
18.26 Introduction to Web Forms Server Controls 584
18.27 HTML Server Controls 585
18.28 Features of HTML server 585
18.29 ASP.NET Server Controls 586
18.30 Adding HTML Server Controls to a Web Forms Page 586
18.31 Adding HTML Server Controls using ASP.NET Syntax 587
18.32 Positioning HTML Elements in Design View 591
18.33 Adding Controls to a Web Forms Page Programmatically 592
18.34 Web Forms Page Processing 593
18.35 The Life Cycle of a Web Forms Page 593
18.36 Stages in Web Forms Processing 595
18.37 Creating a Web Forms Application 596
18.38 Creating Web Forms 604
18.39 Web Services Infrastructure 604
18.40 Visual Studio Enterprise Applications 605
18.41 SOAP 606
18.42 Sending an SMTP Email in HTML Format 612
18.43 Sending an HTML Web Page as an SMTP Email 619

19 UNSAFE CODE AND SECURITY

19.1 Unmanaged Code 627
19.2 The COMImport attribute 629
19.3 Fixed 630
19.4 Security 632
19.5 Modifying Security Permissions in C# 633
19.6 Security and Performance 637

20 Differences between C++ and C# 641

20.1 Simplicity 641
20.2 Consistency 642
20.3 Modernity with Safety 642
20.4 Object Oriented 643
20.5 Type Safety 643
20.6 Scalability 644
20.7 Version Support 644
20.8 Compatibility 645
20.9 Flexibility 646
20.10 Visual Basic.NET 647

20.11 A History of Language Innovation 647
20.12 Overloading 648
20.13 Parameterized Constructors 648
20.14 Free Threading 649
20.15 Structured Exception Handling 649
20.16 Strict Type Checking 650
20.17 Shared Members 650
20.18 Initializers 650
20.19 Conclusion 651
20.20 A Comparative Overview of C# 651
20.21 Background 651

21 Interfacing With Com

21.1 Introduction 669
21.1.1 What We Will Build? 670
21.1.2 Creating Plug-ins 670
21.1.3 The Client Application 676
21.1.4 Background 684
21.1.5 Migrating to COM+ 684
21.1.6 Managing Components 685
21.1.7 Interface Implementation 686
21.1.8 Implements IObjectConstruct 686
21.1.9 Click the Activation tab 686
21.1.10 More Settings 687
21.1.11 Creating a Distributed Application 687
21.1.12 Process for Creating a Distributed Application 688
21.1.13 Create the Middle-Tier Business Object 689
21.1.14 Create and configure a DataSet 690
21.1.15 Expose the DataSet from Your Business Object 691
21.1.16 Create the User Interface 692
21.1.17 Windows User Interface Option 692
21.1.18 Web User Interface Option 693

22 MISCELLANEOUS C# OBJECTS

22.1 Printing in C# 697
22.2 Print Preview 699
22.3 Other Editor Functions 699
22.4 C#.ListView 699
22.5 colorCode 703

23 Automatic memory management using C#

23.1 Memory Management 709
23.2 Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework 711
23.3 Allocate memory for the type that represents the resource 712
23.4 Resource Allocation 712
23.5 The Garbage Collection Algorithm 713
23.6 Finalization 716
23.7 Finalizable objects take longer to allocate 716
23.8 Finalization Internals 718
23.9 Resurrection 721
23.10 Forcing an Object to Clean Up 722

Introduction and Overview of C#

1.1 NET Programming Language

The C# (C Sharp), is an object-oriented programming language announced by Microsoft Corp. on 26.6.2000. It renders itself to a wide range of applications;from the most basic ones to those that can exploit the power of XML-based Web services on the .NET platform.

It is easy to write, read, and maintain like Visual Basic, and offers the power and the flexibility of C++. Provision for type-safety, garbage collection, simplified type declarations, versioning and scalability support, and such other features of C# make developing solutions faster and easier, especially for COM+ and Web Services. This chapter introduces the reader to C#, the language that is all set to dominate the future.

C# language enables quick and easy building of solutions for the Microsoft .NET platform. The framework provided allows C# components to become Web services that are available across the Internet, from any application running on any platform.

The language as such enhances developer productivity and also serves to eliminate programming errors. C# equips the C and C++ programmer with the scope for Web development without curtailing the power and the flexibility that those developers call for.

At the time of writing this book Microsoft is working with ECMA, an international standards body, to create a standard for C#, that will enable multiple vendors to deliver the language and supporting tools.Simplification of Integration and interoperability are thus, at the threshold.

1.2 Evolution of C#

In the past two decades, C and C++ were the most widely used languages for developing commercial and business software. Though both these languages offer a tremendous amount of fine-grained control, this flexibility is acheived at the cost of productivity. Besides, C and C++ applications often take longer to develop as compared to, say, Microsoft Visual Basic. Due to the complexity and long cycle times associated with the C and C++ languages, programmers have been cherishing a language that strikes the right balance between power and productivity. There are languages that enchance the productivity of C and C++, but at the cost of their flexibility. Such solutions constrain the developer too much (say, by omitting a mechanism for low-level code control) and provide least-common-denominator capabilities. Also, they don't easily interoperate with pre-existing systems, nor do they suit current Web programming practices.

C and C++ programmers wanted rapid development combined with power to access all the functionality of the underlying platform. They required an environment that is completely in sync with emerging Web standards and one that provides easy integration with existing applications. Additionally, these developers desired the provision to code at a low level as and when the need arose.

1.3 Features of C#

Being an object-oriented design, C# is gaining popularity with developers who build a wide range of high-performance Web applications and components — from XML-based Web services to middle-tier business objects and system-level applications. The following features are bound to make it the first preference of programmers in due course:

1. C# is a simple yet powerful programming language for writing enterprise applications.
2. The C# language is a hybrid of C and C++. It retains many C++ features in the areas of statements, expressions, and operators.
3. C# introduces considerable improvement and innovations in areas such as type safety, versioning, events, and garbage collection.
4. C# has been crafted to help developers accomplish more with fewer lines of code, and with fewer chances for error.
5. C#, helps developers to easily build Web services that can be used across the Internet — from any language, on any platform.

It is an elegant, simple, type-safe language that offers much to the enterprise programmer in building a variety of applications.

1. While offering enhanced productivity, C# also enables complete access to the underlying platform, and low-level code control, giving developers the power to build complex business systems.
2. Best of all, C# builds on the skills that many programmers and organizations have already developed.This directly increases the productivity and effects time-saving.
3. C# provides access to the common API styles: .NET, COM, Automation, and C-style APIs. It also supports unsafe mode, where you can use pointers to manipulate memory that is not under the control of the garbage collector.
4. C# also enables one to build durable system-level components by virtue of the following features:
   - Full COM/Platform support for existing code integration.
   - Robustness through garbage collection and type safety.
   - Security provided through intrinsic code trust mechanisms.
   - Full support of extensible met a data concepts.
5. C# provides for interoperating with other languages, across platforms, with legacy data, by virtue of the following features:
   - Full interoperability support through COM+ 1.0 and .NET services with tight library-based access.

- XML support for web-based component interaction. Versionability to provide ease of administration and deployment.

Microsoft has submitted the C# language specifications to ECMA for standardization. "*ECMA, a leading international standards body for information technology and corporate telecommunications, is pleased to consider the proposal from Microsoft to develop the standards for a new Internet programming language that can be implemented by vendors to run on any platform or device*," said Jan Van Den Beld, secretary-general of the Geneva-based ECMA. "*We see excellent synergy between this proposal and our existing ECMA Script Standard* (ECMA-262), *which provides the scripting language for the World Wide Web. We consider it of critical importance that this technology follows a truly open standardization approach to ensure interoperability in an interconnected world.*"

The C# language reference document, submitted to ECMA, is available online at http://msdn.microsoft.com/vstudio/nextgen/. C# will be included in the next generation of the Visual Studio development system, and will be available in beta release later this year.

1.4 Benefits of Changing Over to C#

1.4.1 C# is a part of .NET Platform

C# is a modern, object-oriented language that enables programmers to quickly build a wide range of applications for the new Microsoft .NET platform, by providing tools and services that fully exploit both computing and communications. C# includes built-in support to turn any component into a Web service that can be invoked over the Internet—from any application running on any platform. Moreover, the Web services framework can make existing Web services appear like native C# objects to the programmer.Thus developers can avail the object-oriented programming skills they already have, to enhance their existing Web applications. There are some more subtle features that make C# a great Internet programming tool. For instance, XML is emerging as the standard way to pass structured data across the Internet. Such data sets are often very small. For improved performance, C# allows the XML data to be mapped directly into a struct data type instead of a class. This is a more efficient way to handle small amounts of data.

1.4.2 Direct Components for WEB-Services

Its object-oriented design renders C# well-suited for architecting a wide range of components—from high-level business objects to system-level applications. Using simple C# language constructs, these components can be converted into Web services, allowing them to be invoked across the Internet, from any language running on any operating system. The new model solutions use emerging Web standards like Hypertext Mark-up Language (HTML), Extensible Mark-up Language (XML), and Simple Object Access Protocol (SOAP). Existing development tools were developed prior to the Internet or when the Web, as we know it today, was in its infancy. As a result, they don't work well with the latest Web technologies.

1.4.3 RAPID Development Tool

C# is designed to bring rapid development to the programmer without compromising the power and the control that have been the hallmark of C and C++.

C# has much in common with C and C++. Developers familiar with these languages can easily accustom to C#.

1.4.4 Emphasis on Productivity and Safety

The new Web economy—where competitors are just one click away—is taxing on business to respond to competitive threats faster than ever before. Developers are called upon to shorten cycle times and produce more incremental revisions of a program, rather than a single monumental version.

1.4.5 Eliminates Costly Programming Errors

Even expert C++ programmers can make the simplest of mistakes—forgetting to initialize a variable, for instance—and often these simple mistakes result in unpredictable problems that can remain unidentified for long periods of time. A minor indiscretion can turn out to be extremely expensive.

The modern design of C# eliminates the most common C++ programming errors. For example:

1. Garbage collection relieves the programmer of the burden of manual memory management.
2. Variables in C# are automatically initialized by the environment.
3. Variables are type-safe.
4. The C# language makes it easy for developers to write and maintain programs that solve complex business problems.
5. Reduces ongoing development costs with built-in support for versioning.
6. Updating software components is an error-prone task. Revisions made to the code can unintentionally change the semantics of an existing program. To tackle this problem, C# includes the feature of versioning support in the language. For instance, method over-riding has to be explicit; so it cannot happen inadvertently as in C++ or Java. This helps prevent coding errors and preserve versioning flexibility. A related feature is the native support for interfaces and interface inheritance. These features enable complex frameworks to be developed and evolved over time.

1.4.6 Better Mapping between Business Process and Implementation

For effective business planning, it is imperative to have a close connection between the abstract business process and the actual software implementation. But most language tools lack the provision to link business logic with code. For instance, developers probably use code comments today to identify which classes make up a particular abstract business object.With C#, mapping between business process and implementation is possible.

1.4.7 XML Based Metadata

C# inherits the visual development feature of C++ and C languages.This enables millions of C and C++ developers to use existing skills to rapidly build sophisticated XML-based .NET applications. The C# language supports typed, extensible metadata that can be applied to any object. Domain-specific attributes can be developed and applied to any language element-classes, interface etc. as required by the project being developed. The developer can programmatically examine the attributes on each element, say by writing an automated tool that will ensure that each class or interface is correctly identified as part of a particular abstract business object, or by creating reports based on the domain-specific attributes of an object. The close match between the custom metadata and the program code will reinforce the connection between the intended program behaviour and the actual implementation.

1.4.8 Extensive Interoperability

The managed, type-safe environment is adequate for most enterprise applications,yet in practice, some applications do require a "native" code, either for performance reasons or for interoperating with existing application programming interfaces (APIs). This constrains many developers to use C++ even inplace of a more productive development environment. C# addresses this aspect by:

1. Including native support for the Component Object Model (COM) and Windows-based APIs.
2. Allowing restricted use of native pointers.

With C#, every object is automatically a COM object. Developers need not explicitly implement other COM interfaces since the feature is in-built. Similarly, C# programs can natively use existing COM objects, irrespective of the language they are written in.

If the developer opts for it, C# can accomodate a special feature that will enable a program to call any native API. Developers can use pointers and traditional C/C++ features such as manually managed memory and pointer arithmetic inside a specially marked code block. It means that C# programmers can build on their existing C and C++ code base, rather than discarding them and starting afresh. In either case—COM support and native API access—the developer is offered the essential power and control without having to leave the C# environment.

C# is judged the best language from Microsoft for writing COM+ and Windows-based programs for enterprise computing. However, you won't have to migrate your existing C or C++ code.

C# is a simple, modern, object oriented, and type-safe programming language derived from C and C++. C# (pronounced "C sharp") is firmly planted in the C and C++ family tree of languages, and will immediately be familiar to C and C++ programmers. C# aims to combine the high productivity of Visual Basic and the raw power of C++.

C# is provided as a part of Microsoft Visual Studio 7.0. In addition to C#, Visual Studio supports Visual Basic, Visual C++, and the scripting languages VBScript and JScript. All of these languages provide access to the Next Generation Windows Services (NWGS) platform, which includes a common execution engine and a rich class library. The .NET software development kit defines a "Common Language Subset" (CLS), a sort of lingua franca that ensures seamless interoperability between CLS-compliant languages and class libraries. For C# developers, this means that even though C# is a new language, it has complete access to the same rich class libraries that are used by seasoned tools such as Visual Basic and Visual C++. C# itself does not include a class library.

Introduction to C#

C# (pronounced C-Sharp, just like in musical notation) is a new language for Windows applications which is intended as an alternative to the main previous languages, C++ and VB. Its purpose is twofold: It gives access to many of the facilities previously available only in C++, while retaining some of the simplicity of VB. It has been designed specifically with the .NET framework in mind, and hence is very well structured for writing code that will be compiled for .NET. Where Does C# fit In? In programming language terms, if you start from the lowest level of language available to Windows developer and move up, you would see something like this:

```
Assembly Language
    C, C++
        C#, J++
            VB
                VBA, scripting languages
```

As you move down the list, you get to languages that are considered progressively easier to learn and quicker to develop code in, but which leads to less efficient code. Thus, assembly language will give you absolute top-performance code, but it is so hard to write and debug that you will only really find it used in things like selected subroutines in some high-performance games, and some software that controls hardware devices where the performance is critical. For most applications requiring good performance, C++ has traditionally been the language of choice. If you are developing GUI applications and want them written quickly, you probably will opt for VB. Performance, with GUI applications is just a matter of clicking the mouse.

C# is designed to fill a perceived gap between VB and C++. It's aimed at developers who would like to be able to write code more easily than is possible with C++, but without the overhead of VB (A gap that J++ seems to have largely failed to satisfy—although it has to be said that there are a lot of similarities between C# and J++). C# has a syntax that is very similar to C++, and supports many C++ language features, for example inheritance (though not multiple inheritance) of classes. For this reason, C++ programmers are likely to find C# easy to learn.For VB programmers, C# is likely to be easier to learn than C++. C# is also heavily integrated into COM: C# classes are automatically COM objects. (If you don't want that overhead, you can declare your class as a struct instead).

A good example of how C# strikes the right balance between C++ and VB is found in the concept of memory management. In VB, the only memory management you ever need to do is to remember to `Set your objects to Nothing` when you have finished with them —the only penalty if you forget to do this is that the system resources and memory may be used for slightly longer than necessary. On the other hand, C++ not only gives developers the chance to fine tune exactly what variables occupy memory for how long (thus potentially leading to highly efficient memory usage)—it largely requires them to do so. This means C++ programs are prone to memory management bugs, often leading to memory leaks, in a way that can't really happen in VB.

C# mostly takes the VB path on memory management—the language itself takes responsibility for handling it. But unlike VB, if you want to sort out the memory yourself in C#, you can opt to do so by declaring a section of your code as 'unsafe', and then allocating and deallocating memory in the same way as is done in C++.

2

Microsoft's .NET Framework & Platform

2.1. Distributed Computing

The best way to define .NET is to consider as to what it can acheive. At Microsoft, it is believed that the trend is shifting towards distributed computing. Over the last couple of years, people have been laying fat pipes to the point where bandwidth is less limited than it has been in the past. This combined with the Moore's Law effect of processing power doubling every eighteen months with prices being halved, you now have the option to do distributed computing because bandwidth is less expensive.

There are numerous instances of distributed applications today. Napster is an application that uses a rich client talking to a directory service in the cloud and uses all the participating computers on the network as servers. Another example of a distributed application is instant messaging, where you have a rich client that talks to a buddy list in the cloud and communicates with other rich clients — Instant Messenger and Windows in the network. In other words .NET is aiming at accelerating the next generation of distributed computing.

2.2 Three Accelerators of Distributed Computing

The following three aspects are necessary pre-requisites for accelerating the distributed computing generation:

1. **Web Services:** Everything has to be in a Web service. This applies to both software and resources which are stored like a network.
2. **Aggregation and Integration:** Once you have these Web services, you need to be able to aggregate and integrate these Web services in a very simple and easy way.
3. **Simple and Compelling User Experience:** There must be a simple and compelling consumer or end user experience.

2.3 Overview of the .NET Platform

Microsoft .NET(pronounced as dot net), is a Microsoft's XML Web services platform. Its main components are:

1. A software platform for building .NET experiences;
2. A programming model and tools to build and integrate XML Web services; and
3. A set of programmable XML Web services.

Microsoft .NET enables users to interact with a broad range of smart devices through the Web, while ensuring that the user, rather than the application, controls the interaction. Microsoft .NET makes the user experience with applications, services, and devices which are personalized, simple, consistent, and secure. Microsoft .NET is Microsoft's XML Web services platform. .NET contains all that is needed to build and run software based on XML, the lingua franca of Internet data exchange. Microsoft .NET solves several core problems underlying software development today.They are undertaken as follows:

1. Interoperability, integration, and application extensibility are too hard and too expensive. Microsoft .NET's reliance on XML — an open standard language managed by the World Wide Web Consortium (W3C) — removes barriers to data sharing and software integration.
2. The integration challenges are compounded by an avalanche of competing proprietary software technologies that plague the industry. Microsoft .NET is built on open standards and embraces all programming languages.
3. The end users' experience while using their software is not simple or compelling enough. End users are unable to share data easily among their applications, or act on information when they can access it. XML makes it easy to exchange data, and .NET software gives them the ability to work with the data once it is received.
4. End users don't control their personal information and data when working on the Web, leading to privacy and security concerns. Microsoft .NET provides a set of services that let users manage their personal information and control access to that information.
5. Computer programmers and Web site developers have difficulty providing enough value to users, due at least in part to the fact that their applications and services do not play well with others and act as islands of information that aren't connected.

   Microsoft .Net is designed to enable aggregation of value and services from multiple sites and companies into coherent experiences for users.

Just as MS-DOS and Windows operating systems significantly changed computing, so will .NET. MS-DOS drove the acceptance of personal computers throughout businesses and homes; Windows elevated the graphical user interface to the preferred way of interacting with software, and the graphical user interface made personal computing mainstream. .NET is designed to make XML Web services the mainstream model for computing to move forward.

XML Web services are software modules which are built using XML for data exchange to help applications, services, and devices work together. Sharing data through XML allows them to be independent of each other while simultaneously giving them the ability to loosely link themselves into a collaborating group that performs a particular task.

The easiest way to think of how XML Web services work is to compare them to Lego blocks. Just like Lego blocks, XML Web services are independent units. Legos have a standard method of snapping together and so do XML Web services — through XML messaging. When you snap Legos together, you build an object:say, a house, a boat, a giraffe, or an aeroplane. Similarly, when you snap together XML Web services, you build a software solution that performs a particular task. And, just like the way you use the same Lego block (say, the yellow two-by-three rectangle) as part of many different objects, you can use a single XML Web service in many different groups, as part of the solution to many different tasks.

XML Web services also make it possible for developers to choose between building and buying the pieces of their applications, and to choose between consuming other XML Web services to complete their solution or exposing their own services for other applications or services to consume. This means that an individual company doesn't need to supply every piece of a customer solution in order for the customer to have a complete solution. In addition to XML Web services being independent of each other, they are also independent of the device used to access them. Unlike stand-alone applications, XML Web services aren't tied to a particular programming language, business application, or online service. This gives the end users the freedom to work on any access device they choose, from a powerful desktop computer to smart devices like mobile phones and hand-holds. As a result, Microsoft .NET delivers a different type of user experience — a dramatically more personal, integrated experience derived from connected XML Web services, and delivered through the new breed of smart devices.

2.4 Uses of .Net Microsoft

The uses of Microsoft .NET lies in:

1. Usually, end user experience is never simple or compelling. They can not readily share data among their applications or respond to those information. XML renders exchange of data easy and .NET software enables the users to work with the data once they are received.
2. Web-based software have the handicap that end users do not control their personal information and data while working on the web leading to privacy and security concerns.The central services offered by Microsoft.NET let users manage their personal information and control access to such information.

2.5 .NET and its Components

As applications evolve, Microsoft has many tools and technologies designed to make the .NET platform manageable. The current technologies and products include the following:

1. Microsoft Windows 2000 Server to manage directory, events, and distributed clients.
2. Windows 2000 Datacenter Server for guaranteed reliability.
3. Microsoft SQL Server 2000 to manage XML data.
4. Microsoft BizTalk Server to manage events, processes, and contracts across diverse systems.
5. Microsoft Application Center Server to manage clusters.
6. Digital Rights Management (DRM) for managing authorization of content.

7. Universal Description, Discovery, and Integration (UDDI) to promote XML Web services to third parties.

2.5.1 Microsoft's .NET Strategy

.NET programmers write XML Web services, rather than focusing solely on stand- alone applications for servers or clients. They snap together those services into loosely coupled, collaborative constellations of software, using XML messaging to communicate among the XML Web services. To achieve this, programmers need the following:

1. A software platform to build a new type of personal, integrated user experience.
2. A programming model and tools to build and integrate XML Web services.
3. A set of programmable services to provide the foundation for applications and services.

Microsoft's .NET strategy delivers these three things. .NET includes:

1. The .NET Platform, which is a set of programming tools and infrastructure to enable the creation, deployment, management, and aggregation of XML Web services.
2. NET experiences, which are the means for end users to interact with .NET.

2.5.2 The .NET Platform Tools

Microsoft's .NET platform comprises the tools you need to create and run XML Web services. It has four components: (1) .NET Framework and Visual Studio.NET; (2) Server infrastructure; (3) Smart Devices;and (4) .NET experiences.

2.6 .NET Framework and Visual Studio.NET

These are the developer tools to build XML Web services. The .NET Framework is the set of programming interfaces at the heart of the Microsoft .NET platform — Visual Studio. NET is a multi-language suite of programming tools.

2.7 Server Infrastructure

The server infrastructure for .NET, including Windows and the NET Enterprise Servers, is a suite of infrastructure applications for building, deploying, and operating XML Web services. The key technologies include support for XML, scale-out, and business process orchestration across applications and services. These servers include:

1. Application Center 2000 to enable scale-out solutions;
2. BizTalk Server 2000 to create and manage XML-based business process orchestration across applications and services;
3. Host Integration Server 2000 for accessing data and applications on mainframes;
4. Mobile Information 2001 Server to enable use of applications by mobile devices like cell phones; and
5. SQL Server 2000 to store and retrieve structured XML data.

Building Block Services: The building block services are a user- centric set of XML Web services which move the control of user data from applications to users, turning the Web inside out and enabling personalized simplicity and consistency across applications, services, and devices; this while ensuring that the user consent is the basis for all transactions. They include Passport (for user identification) and services for message delivery, file storage, user-preference management, calendar management, and other functions. Microsoft will offer a few building block services in areas that are critical to the infrastructure of .NET; a wide range of partners and developers will significantly expand the set of building block services. You may also locate corporate and vertical building block services built on the .NET platform.

2.8 Smart Devices

.NET uses software for smart devices to enable PCs, laptops, workstations, smart phones, handheld computers, Tablet PCs, game consoles, and other smart devices to operate in the .NET universe. A smart device is:

1. **Smart about you**: It uses your .NET identity, profile, and data to simplify your experience. It is smart about your presence and enables tailoring of notifications in response to presence or lack thereof.
2. **Smart about the network**: It is responsive to bandwidth constraints; provides support for both online and offline use of applications; and understands what services are available.
3. **Smart about information**: It can gain access, analyze, and also act on data anywhere anytime.
4. **Smart about other devices**: It discovers and announces PCs, smart devices, servers, and the Internet; it knows how to provide services to other devices and is smart about accessing information from the PC.
5. **Smart about software and services**: The applications and data are presented optimally for form factor; the input methods and connectivity are appropriate for great end-user interaction; can consume Web services using XML, SOAP, and UDDI; and finally, is programmable and extensible by developers.

Microsoft is working on some of the software for smart devices which include Windows XP, Windows Me, Windows CE, Windows Embedded, the .NET Framework, and the .NET Compact Framework.

2.9 .NET Experiences

The Microsoft .NET experience is a dramatically more personal, integrated computing experience using connected XML Web services when delivered through smart devices. Following are the characteristics of .NET which make it all the more advantageous:

1. It offers a personal and integrated experience. In contrast to silos of information divided by a provider, .NET experiences are centered around the user, integrating their data and preferences into a single application.
2. It has Connected XML Web services.

3. Through the use of XML and Simple Object Access Protocol (SOAP), a range of services tailored to the needs of the user can be fed into a single, integrated experience.
4. Interactions can be delivered through smart devices. The users can experience .NET through their interactions with smart devices. Smart devices are Web enabled appliances, such as personal computers, handheld computers, and smart phones— with software that makes them more intelligent in their interactions with users, the network, information, and other devices, and services. The end users access XML Web services through .NET experiences which are analogous to the current crop of stand-alone applications, but differ from them in a few important ways.

.NET experiences are delivered across multiple devices. Instead of writing a different XML Web service and a different .NET experience for each possible device, .NET experiences can read the characteristics of the device which the end user chooses for access and deliver an appropriate interface.

.NET experiences use XML Web services. .NET experiences take advantage of XML Web services when they are connected to the network, aggregating the additional value for users to solve complete problems.

.NET experiences are user-centric. .NET experiences are focussed on end users, using identity-based building block services for user identification, preferences, notifications, and user data since the user's data is managed by building block services rather than in the application. Users thus are in control of their own data, can ensure its accuracy, and can co-ordinate data between different applications and services.

Microsoft is transitioning four popular products into .NET experiences. Microsoft Office XP is taking the first steps towards providing a .NET experience for knowledge workers. MSN, including MSN Explorer local client, is on the path to creating a consumer-focused .NET experience. The Central small business portal is working to both; provide necessary XML Web services for small business (such as inventory management) and consume important XML Web services (such as eBay). The Visual Studio development system endeavours to provide a .NET experience for developers and to expose MSDN information and company-specific coding guidelines directly in the tools for the developers to use.

2.9.1 What is an Example of a .NET Experience?

A .NET experience is analogous to the user's interaction with a current stand-alone application with three important differences. This can be understood by an example: Imagine you want to turn up the heat in your house while you are away from home. For this, you can use a .NET experience that controls your household utilities through a smart device. The device you use could be your desktop computer while you were at the office, your smart phone while you were driving to the grocery store, or a kiosk while you were shopping for dinner ingredients.

The uniqueness of the .NET experience in this example reveals that each of these devices generate a different interface for the .NET experience. The desktop computer, for example, might present a graphical map and accept mouse input. The smart phone could give you verbal options and ask you to push a button to specify your choice. And the kiosk may have a touch screen and a small set of temperature options. In all these cases, you are using the same .NET

experience to perform the same task, but you see a distinct interface, which is appropriate to the device.

.NET experiences are location-independent. Taking advantage of the strengths of both locally-installed and Web-based applications, .NET experiences can be accessed both online and offline. In the previous example, you could use a .NET experience no matter where you are: at work, in the car, or at the store.

.NET experiences are personalized. .NET experiences read a user's preferences and profile information, stored in building block services, and automatically incorporate the user's information and preferences into what's being presented. In the previous example, when you're turning on the heat, the .NET experience already knows where you live, based on your log-in ID. You don't need to enter an address.

2.10 Microsoft .NET Benefits

Microsoft .NET provides many benefits to programmers, business leaders, IT departments and consumers. They are as follows:

1. Microsoft .NET makes programming easier by maximizing the returns on programming investments. Developers can build reusable XML Web services instead of monolithic applications — they are easier to write and debug. XML Web services collaborate through XML messaging, but are independent, so modifying one won't break another. This is because XML Web services can be part of many .NET experiences and updating a module effectively upgrades all its related .NET experiences. And since XML Web services can be written in virtually any programming language (including C, C++, Visual Basic, COBOL, Perl, Python, and Java, among others), programmers can use the languages they are most productive in while retaining the ability to debug across services or components written in different programming languages.
2. Microsoft .NET reduces the amount of code your programmers need to write. One XML Web service works with all devices, eliminating the need to write a different version for every device. Uncoupling the display characteristics from the .NET experience makes it easy to add new interface technologies, like speech and handwriting recognition, without a need to rewrite the application.
3. Microsoft .NET creates new business models by allowing a company to commercialize its expertise in new ways. For example, a telecommunications company can expose access to voicemail and caller ID as XML Web services, that would allow users to get to them from inside an instant messaging application, email, or another messaging aggregator of the user's choice. Technology vendors can migrate their current software packages to become XML Web services, then sell those services to third parties who want that functionality, or to .NET experience vendors building a new software package.
4. Microsoft .NET allows IT departments to tap other vendors' XML Web services for expertise and outsourced services, reducing the internal costs and expanding the capabilities they can deliver to their customers.
5. Microsoft .NET redefines the term 'user friendly'. End users can move around an intelligent, personalized Internet, which remembers their preferences and delivers the appropriate data at the appropriate time to any smart device they choose.

2.11 How Microsoft .NET Changes Computing

Microsoft .NET will fundamentally change the way we think of and use our computers. Right now two concepts — the server and the Desktop — dominate computing. Since Microsoft .NET is a Distributed-computing paradigm, it does not use the traditional distinction between desktop and server. Instead, processing takes place wherever it makes the most sense, whether that is on a server, a PC, a handheld, or other smart device. This is smart computing.

.NET's computing model affects both businesses and end users, but in different ways. For end users, the changes will produce unparalleled access to a much more personal, integrated computing experience. With businesses, it changes the way they build software and sell products, making IT a significant contributor to corporate success and introducing new business models.

2.11.1 Changes for End Users

Let us now consider an example of how a .NET experience might work for an end user. Mr.X, an anxious business traveler, gets off the plane in London and realizes that he has forgotten his Smart Phone. Without it, he has no clue where to go for his dinner meeting, the phone numbers of the people he is supposed to meet at dinner, nor gain access to all the important documents that he was going to review before this crucial meeting. He rents another Smart Phone from the dealer at the airport, activates it with his Smart Card, and waits for a few seconds while the phone downloads all the appropriate information via its built-in Internet connection. He consequently gets access to all his information — not just his calendar and phone book, but all the files that he can normally access on his PC.

While leaving the airport, Mr.X suddenly trips while getting off the escalator and twists his ankle badly. He tells the phone to call Dr. A's office, where he gets Ms.B, the receptionist. After confirming his identity on the phone, he gives Ms.B permission to access his location and information so that she can find a clinic near him. Ms.B thus determine which clinic has openings, how far they are from him, and whether it is covered by his insurance. After Mr.X gives her permission by touching the appropriate buttons on the Smart Phone, she makes an appointment. After getting off the phone with Ms.B , Mr.X uses the Smart Phone to access a taxi service by having the phone search for the closest taxi, hailing it, and confirming his destination. The driver automatically receives his destination, so all Mr.X has to do is get into the taxi and confirm payment by touching the screen on his Smart Phone.

From the user's point of view, .NET offers significant benefit over today's choice between stand-alone applications or pure Web sites. XML Web services may perform traditional software functions, such as creating documents, calculating numbers, and storing data. But an XML Web service also can be used to provide a service in the offline sense, like calling a taxi. Hailing a cab is not a piece of software churning within a CPU. The service is getting picked up and taken to your destination. The XML Web service that enables this is the software — software gives access, accepts the request, notifies the driver of the right address and arranges for payment — but what the consumer is paying for is getting from one point to another. In both, the case of a more computing-oriented XML Web service and the enabling XML Web service, the end users receive a dramatically more personal, integrated experience, along with the benefits of convenience and ease-of-use available through smart devices.

2.11.2 Changes for Enterprises

Mr.X,reaching home from his trip, needs to submit his expense report. Using his PDA, he identifies himself and views the list of charges on his corporate credit card. He marks the charges related to his London trip for payment. The pharmacy charges, for the painkillers and the Aircast he wore home, Mr.X marks these as personal expenses. He signs off and lets the credit-card company generate the necessary bills. Since he has marked personal charges, the credit-card company looks up his profile and generates a bill that goes to Mr.X personally, using the method he has specified — in this case, Mr.X prefers a direct withdrawal from his checking account. But he also has requested a physical copy of his statement with the pharmacy charges on it so that he has documentation on the medical expense for his taxes.

Based on an option he selected while checking the charges, the credit-card company sends him a PDF document attached to an e-mail; all Mr.X has to do is to print it out. For the business charges, the credit-card company sends an electronic invoice to the company. The bill arrives at the accounting department and Mr.D, the accounts payable clerk, handles it. Upon receiving an e-mail automatically generated by the arrival of the invoice, he logs into the accounting system and opens the invoice. He double-checks the charges and sees that the employee, Mr.X, has okayed them. He initiates a payment. This authorizes an electronic funds transfer between the company's bank and the credit-card company.

From the enterprise's point of view, .NET can handle many tasks automatically, freeing up an employee's time. By linking systems and XML Web services through XML, data exchange and processing becomes significantly easier. In this example, the employees, Mr.X and Mr.D, only come into the equation once each — for performing a one-click approval and initiating a transaction. What they do not do here is to spend time filling in expense reports or manually entering forms data into the accounting system to cut a check.

For enterprises and enterprise end users, .NET promises remarkable services.The services are highly personalized, integrated applications which are derived from connected XML Web services, along with the flexibility delivered through smart devices.

2.11.3 What Stays The Same

Even though Microsoft .NET brings about some radical changes in computing, many things will still continue to remain the same. Following is a list of things which will remain immune to changes:

1. End users will still work with familiar interfaces within .NET experiences like Microsoft Office. This reduces retraining costs and means that end users can begin using .NET-enabled software immediately.
2. Hardware will still run operating systems like Windows, UNIX, Windows CE, and PalmOS. In fact, .NET increases the number of places where software can be running while simultaneously reducing the development burden. Because XML Web services only communicate with devices through XML, any smart device can use an XML Web service.
3. Developers will still use their preferred programming language. The .NET platform, through the .NET Framework's Common Language Runtime, enables XML Web services to interoperate whatever their source language. Further, .NET experiences can be built out of

XML Web services written in Visual Basic, Java, or even COBOL. This neutrality means that no one has to rip and replace current assets to participate in the .NET universe.

4. Legacy systems do not need to be replaced. Some of Microsoft's .NET products are specifically designed to make it simple to integrate existing assets into new .NET XML Web services and .NET experiences. Host Integration Server, for example, simplifies access to mainframes, and BizTalk Server manages business process orchestration that includes existing systems and data formats, while making the necessary automated conversions of data to XML.

This next generation of distributed computing builds on the current generation. Microsoft .NET isn't a wholesale replacement of software applications as we know it, but rather a natural evolution that will bring the benefits of collaboration and interoperability to the isolated technology islands we now have.

2.12 Next Generation of Computing

Microsoft .NET is Microsoft's XML Web services platform. This is the next generation of Internet computing, using XML to communicate among loosely coupled XML Web services that are collaborating to perform a particular task. Microsoft's .NET strategy delivers a software platform to build new .NET experiences, a programming model and tools to build and integrate XML Web services, and a set of programmable Web interfaces. The transition to .NET is happening now. Microsoft has announced the first parts of the .NET Platform — the .NET Framework, Visual Studio .NET, and several other building block services and also the first .NET experiences. Microsoft will continue developing more tools and services every year. Early adopters are working on XML Web services for release in 2001. You can take your first steps towards .NET today. Beta versions of Visual Studio.NET and the .NET Framework, is initially being tested the total number of beta testers of both tools to 750,000.

2.13 Visual Studio.NET

Visual Studio.NET represents a complete development environment in the .NET Framework. This allows programmers to use the languages they know and are comfortable with, to make exciting, reusable XML Web services. And it is done by using BizTalk Server, SQL Server, and Windows 2000 Server which further supports your ability to host and run XML Web services by integrating XML on a deep level. By integrating Passport into existing Web applications, software architects can improve current users' experiences and prepare for future gains that .NET promises in ease of use through intelligent, secure identity recognition. Visual Studio.NET version 7.0 is a complete set of development tools for building Web applications, Web Services, and desktop applications. Visual Basic, Visual C++, and C# use the same integrated development environment (IDE), which allows them to share tools and facilitates in the creation of mixed-language solutions.

2.14 The .NET Framework

The .NET Framework is an environment for building, deploying, and running Web Services and other applications.It consists of three main parts:the Common language Runtime; the Framework classes; and the ASP.NET.

The .NET Framework enables you to create great Web applications. However, it can also help you build the same applications you build today. If you write any Windows software (using ATL/COM, MFC, Microsoft Visual Basic, or even standard Microsoft Win32, .NET offers many advantages to the way you currently build applications. Of course, if you develop Web sites, then the .NET Framework has a lot to interest you—starting with ASP.NET. The .NET Framework is a multi-language environment for building, deploying, and running Web Services and applications. It consists of three main parts:

(1) Common Language Runtime (CLR)
Despite its name, the runtime actually has a role in both; a component's runtime and development time experiences. While the component is running, the runtime is responsible for managing memory allocation, starting up and killing threads and processes, and enforcing security policy, as well as satisfying any dependencies that the component might have on other components. At development time, the runtime's role changes slightly; because it automates so much (for example, memory management), the runtime makes the developer's experience very simple, especially when compared to COM today. In particular, features such as reflection dramatically reduce the amount of code a developer must write in order to turn business logic into a reusable component.

(2) Unified Programming Classes
The frameworks provide developers with a unified, object-oriented, hierarchical, extensible set of class libraries (APIs). Currently, C++ developers use the Microsoft Foundation Classes, Java developers use the Windows Foundation Classes, and Visual Basic developers use Visual Basic's APIs. The frameworks unify these disparate frameworks. By creating a common set of APIs across all programming languages, the .NET Framework enables cross-language inheritance, error handling, and debugging. All programming languages, from JScript to C++, become equals and developers are free to choose the language that they want to use.

(3) Active Server Pages.NET
ASP.NET builds on the .NET Framework's programming classes, providing a Web application model in the form of a set of controls and infrastructure that make it simple to build Web applications. Developers are exposed to a set of ASP.NET controls that encapsulate common HTML user interface elements such as text boxes, drop-down menus, and so on. These controls actually run on the Web server, however, and simply project their user interface as HTML to a browser. On the server, the controls expose an object-oriented programming model that brings the richness of object-oriented programming to the Web developer. ASP.NET also provides infrastructure services such as session state management and process recycling that further reduce the amount of code a developer must write and increase application reliability. In addition, ASP.NET uses these same concepts to enable developers to deliver software as a service. Using ASP.NET Web Services features, ASP.NET developers can write their business logic and the ASP.NET infrastructure will be responsible for delivering that service via SOAP.

2.15 Visual Studio .NET Highlights

This section provides information about some of the latest features available in the release of Visual Studio Language Enhancements:

(1) Visual Basic
Visual Basic has been updated to include many new and improved language features that make it a powerful object-oriented programming language. These features include inheritance, interfaces, and overloading, among others. Visual Basic now also supports structured exception handling, custom attributes, and Common Language Specification (CLS). In addition, Visual Basic now supports free threading. Free threading is the ability to assign individual tasks to separate processing threads.

(2) C#
C#, pronounced C Sharp, is a new object-oriented programming language that is an evolution of C and C++, providing a simple and type safe language for developing enterprise applications.

(3) VC++
Visual C++ Managed Extensions for C++ and attributed programming, along with the introduction of C#, are just some of the enhancements made to the C++ language. Managed Extensions simplify the task of migrating existing C++ applications to the new .NET framework. Attributes, like C++ keywords, are used in your source files and interpreted by the compiler. Attributes are designed to provide a quick and efficient method to simplify COM programming with Visual C++.

(4) Web Forms
Web Forms are an ASP.NET technology that you use to create programmable Web pages. Web Forms use pure HTML, which allows any browser on any platform to view the pages. Using Web Forms, you create Web pages by dragging and dropping controls onto the designer and then add the code— similar to the way that you create Visual Basic forms.

(5) Web Services
Web Services are applications that can receive requests and data using XML over HTTP. Web Services are not tied to a particular component technology or object-calling convention and can therefore be accessed by any language, component model, or operating system. In Visual Studio.NET, you can quickly create and include Web Services using Visual Basic, C#, or ATL Server.

(6) XML
Extensible Markup Language (XML) provides a framework for describing structured data. XML is a subset of SGML that is optimized for delivery over the Web. The World Wide Web Consortium (W3C) defines XML standards so that structured data will be uniform and independent of applications. Visual Studio.NET fully supports XML, providing the XML Designer to make it easier to edit XML and create XML schemas.

2.16 Getting Started

You can use several methods to become familiar with this release of Visual Studio.NET. These are:

1. Visual Studio Walkthroughs guide you through the completion of a task using Visual Studio.NET. Use walkthroughs to become familiar with the product and see how the various technologies can be used together.
2. Rich Client Application Describes the process for building a dialog-based application using several control types, connecting to an existing database, and then displaying the data using a data grid and other controls. You can choose to view the walkthrough tailored for C# or Visual Basic, Microsoft Foundation Class (MFC) Library, Managed Extensions for C++, or Native C++ upgrade.
3. Web Forms Application describes the process for creating a Web-based application integrated with business-object components written for both C# and Managed Extensions for C++.
4. Creating and Accessing Web Services Describes the process for creating a Web service using Visual Basic or C#, or ATL Server, and then using the Web service from an application.
5. Enterprise Templates: Initial Structure and Advantages of Using Enterprise Templates to create Distributed Applications.
6. Migrating Existing Applications: As with each release, some tools and technologies are replaced with better methods for achieving your programming goals. As a result, applications created using an older version of Visual Studio may need to be updated in order to load and build correctly using the current version of Visual Studio.

2.17 Common Language Runtime (CLR)

The .NET Framework provides a run-time environment called the Common Language Runtime, which manages the execution of code and provides services that make the development process easier. Compilers and tools expose the runtime's functionality and enable you to write code that benefits from this managed execution environment.

Code that you develop with a language compiler that targets the runtime is called managed code; it benefits from features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services.

To enable the runtime to provide services to managed code, language compilers are required to emit metadata, which means that they provide information that describes the types, members, and references in your code. Metadata is stored along with the code: every loadable common language runtime image contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set up run time context boundaries.

The runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. Objects whose lifetimes are managed in this way by the runtime are called managed data. Automatic memory management eliminates memory leaks as well as some other common programming errors. If your code is managed, you can use managed data; however, instead of that you can use unmanaged data if you wish, or you can use both managed and unmanaged data in your .NET application. Because language compilers supply their own types, such as primitive types, you might not always know (or need to know) whether your data is being managed.

The Common Language Runtime makes it easy to design components and applications whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated. For example, you can define a class, then, using a different language, derive a class from your original class or call a method on it. You can also pass an instance of a class to a method on a class written in a different language. This cross-language integration is possible because language compilers and tools that target the runtime use a common type system defined by the runtime, and they follow the runtime's rules for defining new types, as well as creating, using, persisting, and binding to types. As part of their metadata, all managed components carry information about the components and resources they were built against. The runtime uses this information to ensure that your component or application has the specified versions of everything it needs; as a result, your code is less likely to break due to some dependency not being met. Registration information and state data are no longer stored in the registry where it can be difficult to establish and maintain; instead, information about the types you define (and their dependencies) is stored with the code as metadata, making the task of component replication and removal much less complicated.

2.17.1 Services Provided by CLR

The Common Language Runtime is the execution engine for .NET Framework applications. It provides a number of services, including the following:

1. Code management (loading and execution).
2. Application memory isolation.
3. Verification of type safety.
4. Conversion of IL to native code.
5. Access to metadata (enhanced type information).
6. Managing memory for managed objects.
7. Enforcement of code access security.
8. Exception handling, including cross-language exceptions.
9. Interoperation between managed code, COM objects, and pre-existing DLLs (unmanaged code and data).
10. Automation of object layout.
11. Support for developer services (profiling, debugging, and so on).

Language compilers and tools expose the runtime's functionality in ways that are intended to be useful and intuitive to their developers. This means that some features of the runtime might be

more noticeable in one environment than in another. Therefore, how you experience the runtime depends on which language compilers or tools you use. For example, if you are a Visual Basic developer, you might notice that with the Common Language Runtime, the Visual Basic language has more object-oriented features than before. The following benefits of the runtime might be particularly interesting to you:

1. Performance improvements.
2. The ability to easily use components developed in other languages.
3. Extensible types provided by a class library.
4. A broad set of language features.

If you use Visual C++, you can write managed code using managed extensions to C++, which provide the benefits of a managed execution environment while giving you access to powerful capabilities and expressive data types that you are familiar with. You might find the following runtime features especially compelling:

1. Cross-language integration, especially cross-language inheritance.
2. Automatic memory management, which manages object lifetime so that reference counting is unnecessary.
3. Self-describing objects, which make using IDL (Interface Definition Language) unnecessary.
4. The ability to compile once and run on any CPU and operating system that supports the runtime.

2.17.2 In-process and Cross-process Communication Work in the Common Language Runtime

There are two aspects to in-process communication:

1. Between contexts within a single application domain, or across application domains.

   Between contexts in the same application domain, proxies are used as an interception mechanism. No marshaling/serialization is involved. When crossing application domains, we do marshaling/serialization using the runtime binary protocol.
2. Cross-process communication uses a pluggable channel and formatter protocol, each suited to a specific purpose.

If the developer specifies an endpoint using the tool soapsuds.exe to generate a metadata proxy, HTTP channel with SOAP formatter is the default.

If a developer is doing explicit remoting in the managed world, it is necessary to be explicit about what channel and formatter to use. This may be expressed administratively, through configuration files, or with API calls to load specific channels. Options are:

1. HTTP channel w/ SOAP formatter (HTTP works well on the Internet, or anytime traffic must travel through firewalls)
2. TCP channel w/ binary formatter (TCP is a higher performance option for local-area networks (LANs))
3. SMTP channel w/ SOAP formatter (only makes sense cross-machine)

When making transitions between managed and unmanaged code, the COM infrastructure (specifically, DCOM) is used for remoting. In interim releases of the CLR, this applies also to serviced components (components that use COM+ services). Upon final release, it should be possible to configure any remotable component. Distributed garbage collection of objects is managed by a system called "leased based lifetime." Each object has a lease time, and when that time expires, the object is disconnected from the remoting infrastructure of the CLR. Objects have a default renew time i.e. the lease is renewed when a successful call is made from the client to the object. The client can also explicitly renew the lease.

2.18 Common Type System (CTS)

The common type system is a rich type system, built into the Common Language Runtime, that supports the types and operations found in most programming languages. The common type system supports the complete implementation of a wide range of programming languages.

2.19 Common Language Specification (CLS)

The Common Language Specification is a set of constructs and constraints that serves as a guide for library writers and compiler writers. It allows libraries to be fully usable from any language supporting the CLS, and for those languages to integrate with each other. The Common Language Specification is a subset of the common type system. The Common Language Specification is also important to application developers who are writing code that will be used by other developers. When developers design publicly accessible APIs following the rules of the CLS, those APIs are easily used from all other programming languages that target the Common Language Runtime.

2.20 Microsoft Intermediate Language (MSIL)

MSIL is the CPU-independent instruction set into which .NET Framework programs are compiled. It contains instructions for loading, storing, initializing, and calling methods on objects. Combined with metadata and the common type system, MSIL allows for true cross-language integration. Prior to execution, MSIL is converted to machine code. It is not interpreted.

2.21 Managed Code and Managed Data

Managed code is code that is written to target the services of the Common Language Runtime (see What is the Common Language Runtime?). In order to target these services, the code must provide a minimum level of information (metadata) to the runtime. All C#, Visual Basic.NET, and JScript.NET code is managed by default. Visual Studio.NET and C++ code is not managed by default, but the compiler can produce managed code by specifying a command-line switch (/CLR).

Closely related to managed code is managed data_data that is allocated and de-allocated by the Common Language Runtime's garbage collector. C#, Visual Basic, and JScript.NET data is managed by default. C# data can, however, be marked as unmanaged through the use of special keywords. Visual Studio.NET and C++ data is unmanaged by default (even when using the /CLR

switch), but when using Managed Extensions for C++, a class can be marked as managed by using the __gc keyword.

As the name suggests, the __gc keyword means that the memory for instances of the class is managed by the garbage collector. In addition, the class becomes a full participating member of the .NET Framework community, with the benefits and restrictions that brings. An example of a benefit is proper interoperability with classes written in other languages (for example, a managed C++ class can inherit from a Visual Basic class). An example of a restriction is that a managed class can only inherit from one base class.

2.22 Assemblies

An assembly is the primary building block of a .NET Framework application. It is a collection of functionality that is built, versioned, and deployed as a single implementation unit (as one or more files). All managed types and resources are marked either as accessible only within their implementation unit, or as accessible by code outside that unit.

Assemblies are self-describing by means of their manifest, which is an integral part of every assembly. The manifest is as follows:

1. Establishes the assembly identity (in the form of a text name), version, culture, and digital signature (if the assembly is to be shared across applications).
2. Defines what files (by name and file hash) make up the assembly implementation.
3. Specifies the types and resources that make up the assembly, including those which are exported from the assembly.
4. Itemizes the compile-time dependencies on other assemblies.
5. Specifies the set of permissions required for the assembly to run properly.

This information is used at run time to resolve references, enforce version binding policy, and validate the integrity of loaded assemblies. The runtime can determine and locate the assembly for any running object, since every type is loaded in the context of an assembly. Assemblies are also the unit at which code access security permissions are applied. The identity evidence for each assembly is considered separately when determining what permissions to grant the code it contains. The self-describing nature of assemblies also helps makes zero-impact install and XCOPY deployment feasible.

2.22.1 What are Private Assemblies and Shared Assemblies

A private assembly is used only by a single application, and is stored in that application's install directory (or a subdirectory therein). A shared assembly is one that can be referenced by more than one application. In order to share an assembly, the assembly must be explicitly built for this purpose by giving it a cryptographically strong name (referred to as a shared name). By contrast, a private assembly name need only be unique within the application that uses it. By making a distinction between private and shared assemblies, we introduce the notion of sharing as an explicit decision. Simply by deploying private assemblies to an application directory, you can guarantee that the application will run only with the bits it was built and deployed with. References to private assemblies will only be resolved locally to the private application directory.

There are several reasons for which you may build and use shared assemblies, such as the ability to express version policy. The fact that shared assemblies have a cryptographically strong name means that only the author of the assembly has the key to produce a new version of that assembly. Thus, if you make a policy statement that says you want to accept a new version of an assembly, you can have some confidence that version updates will be controlled and verified by the author. Otherwise, you don't have to accept them.

For locally installed applications, a shared assembly is typically explicitly installed into the global assembly cache (a local cache of assemblies maintained by the .NET Framework). Key to the version management features of the .NET Framework is that the downloaded code does not affect the execution of locally installed applications. Downloaded code is put in a special download cache and is not globally available on the machine even if some of the downloaded components are built as shared assemblies. The classes that ship with the .NET Framework are all built as shared assemblies.

Now, if we want to build a shared assembly, does that require the overhead of signing and managing key pairs? Building a shared assembly does involve working with cryptographic keys. Only the public key is strictly needed when the assembly is being built. Compilers targeting the .NET Framework provide command line options (or use custom attributes) for supplying the public key when building the assembly. It is common to keep a copy of a common public key in a source database and point build scripts to this key. Before the assembly is shipped, the assembly must be fully signed with the corresponding private key. This is done using an SDK tool called SN.exe (Strong Name).

Strong name signing does not involve certificates like Authenticode does. There are no third party organizations involved, no fees to pay, and no certificate chains. In addition, the overhead for verifying a strong name is much less than it is for Authenticode. However, strong names do not make any statements about trusting a particular publisher. Strong names allow you to ensure that the contents of a given assembly haven't been tampered with, and that the assembly loaded on your behalf at run time comes from the same publisher as the one you developed against. But it makes no statement about whether you can trust the identity of that publisher.

2.22.2 Difference between a Namespace and an Assembly Name

A namespace is a logical naming scheme for types in which a simple type name, such as MyType, is preceded with a dot-separated hierarchical name. Such a naming scheme is completely under the control of the developer. For example, types `MyCompany.FileAccess.A` and `MyCompany.FileAccess.B` might be logically expected to have functionality related to file access.

The .NET Framework uses a hierarchical naming scheme for grouping types into logical categories of related functionality, such as the ASP.NET application framework, or remoting functionality. Design tools can make use of namespaces to make it easier for developers to browse and reference types in their code. The concept of a namespace is not related to that of an assembly. A single assembly may contain types whose hierarchical names have different namespace roots, and a logical namespace root may span multiple assemblies. In the .NET Framework, a namespace is a logical design-time naming convenience, whereas an assembly establishes the name scope for types at run time.

2.23 Interoperability of COM Objects and .NET

Framework program Any COM component you have deployed today can be used from managed code, and in common cases the adaptation is totally automatic.

Specifically, COM components are accessed from the .NET Framework by use of a runtime callable wrapper (RCW). This wrapper turns the COM interfaces exposed by the COM component into .NET Framework-compatible interfaces. For OLE automation interfaces, the RCW can be generated automatically from a type library.

For non-OLE automation interfaces, a developer may write a custom RCW and manually map the types exposed by the COM interface to .NET Framework-compatible types. Similarly any managed types you build today can be made accessible from COM, and in the common case the configuration is totally automatic. There are certain new features of the managed development environment that are not accessible from COM. For example, static methods and parameterized constructors cannot be used from COM.

In general, it is a good idea to decide in advance who the intended user of a given type will be. If the type is to be used from COM, you may be restricted to using those features that are COM accessible. Depending on the language used to write the managed type, it may or may not be visible by default. Specifically, .NET Framework components are accessed from COM by using a COM callable wrapper (CCW). This is similar to an RCW, but works in the opposite direction. Again, if the .NET Framework development tools cannot automatically generate the wrapper, or if the automatic behavior is not what you want, a custom CCW can be developed.

2.24 How scalable is .NET?

Through the use of .NET server technology—the ability to reuse and update XML Web services on the fly as well as distributing computing power across multiple devices—.NET offers a highly scalable platform.

1. Scale out with Windows 2000 Server, SQL Server 2000, and Application Center 2000.
2. Scale up with Windows 2000 Server and SQL Server 2000.

Use smart devices and XML Web services to break the linear scaling dynamic. By distributing computing to the edge of the network (where there is an abundance of computing power), the bottleneck of server processing power can be removed. By making it simple for the services which are the building blocks of applications to be distributed anywhere on the network, other bottlenecks can be eliminated or minimized, as well.

2.25 .NET : Security and Privacy Concerns

.NET provides strong privacy protection by placing users in control of their own data. Users will dictate which people and applications can access or alter particular pieces of their personal data. These permissions are associated with binding privacy policies which outlive specific transactions.

Effective security requires a combination of secure software and good operational practices, physical security, and license agreements. Some of the current and upcoming software security programs and technologies that are part of the .NET platform include:

1. Windows 2000 for Kerberos, the Microsoft Active Directory service, PKI, EFS, and IP Sec.
2. Building block services for simple, private, and secure experiences with full user control of the data they own.
3. XML messaging interfaces on the building block services, which enable auditing and accountability.
4. Digital Rights Management (DRM) for authorization of content.

2.26 .NET Vs XML Platforms

XML is the common language of data exchange on the Internet. It provides a widely-embraced, open standard technology for data exchange and transformation, overcoming barriers to application and service interoperability and integration. The Microsoft .NET platform delivers the following unique benefits to developers:

1. It's easier, faster, and less expensive to build and integrate XML Web services through the .NET Framework and Tools (Visual Studio.NET).
2. The user experience is richer and more compelling because of smart devices and information agents.
3. The .NET experience is personal and manageable because of identity-centric building block services.
4. It's built from the ground-up for the next generation of software with XML at the core.

The difference between an XML Web service and a Web site is explained as follows:

Unlike Web sites, which are pictures of data designed to be viewed in a browser by a person, an XML Web service is designed to be accessed directly by another service or software application. XML Web services are reusable pieces of software that interact programmatically over the network through industry recognized standards (XML and SOAP). XML Web services can be combined with each other and other applications to build intelligent interactive .NET experiences.

2.27 Advantages of .NET Programming Model and Tools

Visual Studio.NET and the .NET Framework offer the following benefits to developers:

1. The best, fastest, and least expensive way to build XML Web services.
2. A highly productive, multi-language environment for building and running XML Web services.
3. The ability to create secure, scalable applications that can run on multiple devices, saving valuable time and resources.

2.28 Application Deployment and Isolation

The .NET Framework simplifies deployment by making zero-impact install and XCOPY deployment of applications feasible. Because all requests are resolved first to the private application directory, simply copying an application's directory files to disk is all that is needed to run the application. No registration is required.

This scenario is particularly compelling for Web applications, Web Services, and self-contained desktop applications. However, there are scenarios where XCOPY is not sufficient as a distribution mechanism. An example is when the application has little private code and relies on the availability of shared assemblies, or when the application is not locally installed (but rather downloaded on demand). For these cases, the .NET Framework provides extensive code download services and integration with the Windows Installer.

The code download support provided by the .NET Framework offers several advantages over current platforms, including incremental download, code access security (no more Authenticode dialogs), and application isolation (code downloaded on behalf of one application doesn't affect other applications). The Windows Installer is another powerful deployment mechanism available to .NET applications. All of the features of Windows Installer, including publishing, advertisement, and application repair will be available to .NET applications in Windows Installer 1.5.

2.29 Application Domain

An application domain (often AppDomain) is a virtual process that serves to isolate an application. All objects created within the same application scope (in other words, anywhere along the sequence of object activations beginning with the application entry point) are created within the same application domain. Multiple application domains can exist in a single operating system process, making them a lightweight means of application isolation. An OS process provides isolation by having a distinct memory address space. While this is effective, it is also expensive, and does not scale to the numbers required for large web servers. The Common Language Runtime, on the other hand, enforces application isolation by managing the memory use of code running within the application domain. This ensures that it does not access memory outside the boundaries of the domain. It is important to note that only type-safe code can be managed in this way (the runtime cannot guarantee isolation when unsafe code is loaded in an application domain).

2.30 Windows in .NET

Microsoft Windows currently powers the ultimate smart device: the personal computer. Microsoft Windows XP will be able to consume XML Web services, such as photo services, as well as utilize XML Web services to publish to the Web. It will act as a hub for smart devices and interact in the next generation of peer-to-peer computing, real-time communications, and media services. Microsoft Windows Whistler Server will include built-in support for XML and SOAP and will ship with the .NET Framework.

2.31 Digital Nervous System

Organizations need the ability to harness information technology to provide a competitive advantage. Microsoft's vision for corporate computing encompasses a corporate Digital Nervous System to provide free flow of information that empowers corporations to react faster to changing business needs, make more informed decisions, get closer to customers and business partners, and focus on business, not technology.

Development tools play a crucial role in enabling corporations to achieve a Digital Nervous System. Microsoft's enterprise development strategy centers on the Visual Studio development tools suite, in conjunction with the Windows Distributed interNet Application (DNA) framework. Together, Visual Studio and Windows DNA enable organizations to rapidly build well-architected solutions that can provide a competitive edge in the market.

2.32 .NET Framework SDK

The development tools in this SDK will utilize the .NET Framework to allow developers to quickly build and deploy robust applications that take advantage of the new .NET Common Language Runtime (CLR) environment. Using the tools with the .NET Framework SDK provides:

1. A fully managed, protected, and rich application execution environment.
2. Application integration with Active Server Pages (ASP).
3. Improved isolation of application components.
4. Simplified application deployment.

This section begins by demonstrating for writing the classic "Hello World" program in two familiar languages :an update to C++ called "Managed Extensions for C++" and Visual Basic— as well as in C# — designed specifically for the .NET environment. This simple program is then greatly expanded to show a small, componentized client/server application where both the client and server are written using each of the three languages. A fourth client program demonstrates by calling these components from the new Windows-based forms library. The final client program calls these components from ASP.NET, the next generation of Active Server Pages for Web-based development. The steps necessary to construct, compile, and run each program are covered in detail. An appendix contains additional information on several useful developer utilities.

2.33 Microsoft .NET Framework SDK Tools & Utilities

The Microsoft .NET Framework SDK tools and utilities provide developers with the core set of functionality needed to develop, configure and deploy applications and components that use the Microsoft .NET Framework technologies. For information on each of the tools, please look in the Microsoft .NET Framework SDK Documentation under .NET Framework Tools and Debugger. The tools & utilities are located in the SDK's \bin directory. Note to Windows 9X

users: Your Path, Include, and Lib environment variables must be modified for the .NET Framework Tools to function properly.

2.33.1 Configuration and Deployment Tools & Utilities

The following are the important points that are unique to the configuration and deployment tools and utilities:

1. Assembly Generation Utility (al.exe) Accepts as input one or more files that are either MSIL format or resource files and outputs a file with an assembly manifest.
2. Global Assembly Cache Utility (gacutil.exe) used to view and manipulate the contents of the global assembly cache. This utility can be used for building scripts, makefiles, and batch files.
3. Installer Utility (InstallUtil.exe) works in conjunction with the Installer Framework. It allows you to install and uninstall server resources with an assembly, by executing the installer components of that assembly.
4. Assembly Registration Tool (RegAsm.exe) Enables classic COM clients to call managed classes. RegAsm reads the metadata within an assembly, then adds the necessary entries to the registry. So classic COM clients can create the managed classes transparently.
5. Services Registration Tool (RegSvcs.exe) Adds managed classes to Windows 2000 Component Services by performing several tasks within a single utility. These tasks include loading and registering the assembly, and generating, registering, and installing the type library into an existing COM+ 1.0 application.
6. Assembly Cache Viewer (shfusion.dll) is the Windows shell extension for viewing and manipulating the contents of the global assembly cache using the Windows Explorer.
7. Isolated Storage Utility (storeadm.exe) is used to manage isolated storage for the currently logged in user.
8. Type Library Exporter (TlbExp.exe) Uses a managed assembly as input, generates a type library containing COM definitions of the public types defined in that assembly.
9. Type Library Importer (Tlbimp.exe) Converts the type definitions found within a COM type library into equivalent definition in managed metadata format.
10. Web Service Utility (WebServiceUtil.exe) Installs and uninstalls managed code Web Services.
11. Common Language Runtime XML Schema Definition Tool (xsd.exe) is used for working with XML Schemas that follow the XML Schema Definition (XSD) language proposed by the W3C.

2.33.2 Windows Forms Design Tools & Utilities

The Windows Forms ActiveX Control Importer (aximp.exe) uses an ActiveX control's type library as input and generates a wrapper control that allows the ActiveX control to be hosted by a Windows Forms form. Following are the Windows Forms Design Tools & Utilities:

1. License Compiler (lc.exe) Using text files containing licensing information as input, produces a binary .licenses file that can be embedded in a managed binary executable.
2. Resource File Generator Utility (ResGen.exe) Reads text files containing name/value pairs produces a managed binary .resources file.

3. Windows Forms Designer Test Container (windes.exe) is used for testing the design time behaviour of Windows Forms controls.

2.33.3 Security Tools & Utilities

Following are the Security Tools and Utilities that are utilized herein:

1. Code Access Security Policy Utility (caspol.exe) is used to examine and modify machine and user code access security policies.
2. Software Publisher Certificate Test Utility (cert2spc.exe) creates— for test purposes only— a Software Publisher's Certificate (SPC) from one or more X.509 certificates.
3. Certificate Manager Utility (certmgr.exe) is used to manage certificates, certificate trust lists (CTLs), and certificate revocation lists (CRLs).
4. Certificate Verification Utility (chktrust.exe) is used to check the validity of a signed file.
5. Certificate Creation Utility (makecert.exe) creates a Software Publisher's Certificate (SPC) from one or more X.509 certificates for test purposes only.
6. Permissions View Utility (permview.exe) is used to view the code requests in an assembly.
7. Secutil Utility (SecUtil.exe) extracts the Strong Name public key information or Authenticode(tm) publisher certificates from an assembly, in a format that can be incorporated directly into code.
8. Set Registry Utility (setreg.exe) is utilized for changing the registry settings related to test certificates.
9. File Signing Utility (signcode.exe) is used to sign a PE file with requested permissions to give developers more detailed control over the security restrictions placed on their component.
10. Strong Name Utility (Sn.exe) creates and verifies assemblies with strong names.

2.33.4 General Tools & Utilities

Common Language Runtime Debugger (cordbg.exe) is a command-line utility that can help tools vendors and application developers find and fix bugs in programs that target the Common Language Runtime. Following are the other general tools and utilities:

1. Common Language Runtime Visual Debugger (DbgUrt.exe) is a basic debugger with a graphical interface, and is located in the SDK's GuiDebug sub-directory. Documentation for this debugger can be found in the SDK documentation under ".NET Framework Tools and Debugger." Note: The Common Language Runtime Visual Debugger is not installed on Windows 9X platforms.
2. Common Language Runtime IL Assembler (ilasm.exe) Using IL as input, generates a PE file containing the IL and the required metadata. The resulting executable can run to determine whether the IL performs as expected.
3. Microsoft .NET Framework IL Disassembler (ildasm.exe) is used to disassemble and inspect the IL code in a PE file.

4. PEVerify Utility (peverify.exe) Assists in validating the type safety of code prior to release. Also assists in generating IL, and in determining if IL code and associated metadata meets type safety verification requirements.
5. Soapsuds (soapsuds.exe) Converts CLR MetaData to and from XML Schema. Fully represents the full CLR type system needed for CLR Remoting.
6. Windows Forms Class Viewer (wincv.exe) Finds the managed classes matching a specified search pattern, which uses the reflection API to display information about those classes.

2.34 Tools Required

In general, the .NET Framework SDK includes everything necessary to compile and test the samples that accompany this document. In particular, the SDK comes with command-line compilers for each of the languages, the runtime and associated files, and detailed documentation. If you intend to run the ASP.NET sample, you must install the .NET Framework SDK after installing IIS to recognize and properly handle .aspx files (this may require reinstalling the SDK). All of the files needed to use the accompanying samples and start developing are installed with the SDK. The documentation (in the form of compiled HTML, or .chm files) is located in the Docs subdirectory of the SDK. The .NET Framework runtime files and the three language compilers are found in a directory under the Windows directory:

```
\%windir%\Microsoft.NET\Framework\v1.0.????
```

where ???? is the final build number. For Beta 1 of the .NET Framework SDK, the important files are:

(I) mscorlib.dll
(II) bc.exe
(III) cl.exe
(IV) csc.exe

Since source and project files for all three languages are plain text, any text editor - even Notepad - is adequate for examining and modifying the accompanying sample files. However, if you want to make substantial changes to the samples — for instance, creating a new application based on the sample — you will likely want to obtain and install Beta 1 of Visual Studio .NET. The .NET Framework SDK includes a variety of samples, several of which are useful to developers: These are described in Appendix A: "Exploring Namespaces", in this document.

To install and build these SDK samples, please refer to the instructions in StartSamples.htm in the \Samples subdirectory of the SDK. The sample programs that accompany this section — and other samples— are located in the self-extracting "samples.exe" file. Finally, to compile and run the samples that accompany both the SDK and this tutorial, you must have an environment variable correctly configured: path - Must include the subdirectories where the compilers are located. If you use C++, you will also need to have appropriate include and lib environment variables. All of these should be configured by the setup process for the SDK.

2.35 VB.Net

Visual Basic.NET Offers Developers the Newest Levels of Power, Productivity, Scalability and Reliability. Visual Basic is a programming language that has been around for 10 years, and Microsoft owes much of its success to its more than 3 million developers worldwide who have been writing applications using Visual Basic. Just as Visual Basic revolutionized the way in which developers created GUI (graphical user interface) applications, Visual Basic.NET will revolutionize the way user develop next-generation applications for Windows and the Web. Visual Basic.NET is going to transform the millions of VB developers today into the hottest commodity on the job market by giving them the most powerful and versatile Visual Basic to create enterprise-critical applications.

First, Visual Basic.NET includes a host of new features that enable developers to maximize their productivity. In addition to a new unified IDE (integrated development environment), VB developers can take advantage of the new Server Explorer to easily integrate server-side components into their apps, the shared Task List to help organize code, Dynamic Help for instant access to pertinent information, and Macros to customize, extend, and integrate the Visual Studio environment. Maximizing productivity was a key design goal of this product, and with Visual Basic.NET, we've delivered. In addition, the Visual Basic.NET language provides a modernized, simplified syntax that will make developers' code more robust, scalable and flexible. Rather than simply adding some new features to Visual Basic 6.0, Microsoft has re-engineered the product to make it easier than ever before to write distributed applications such as Web and enterprise n-tier systems.

In Visual Basic.NET, we deliver on all of the top requests we've heard from developers over the past few years. This includes first-class object oriented constructs such as implementation inheritance and structured exception handling as well as power features such as free threading. In addition, new graphical designers for developing Windows and Web applications as well as for working with XML and server-side components will make VB developers more versatile than they've ever been in the past.

2.36 SERVLET ENGINES JRUN, ASP AND JSP

The servlet engine manages servlets, maintains HTTP session state using cookies and local data stored on the server, and can store user profiles. Sun, IBM, and Live Software each provide servlet engines that work with all major Web servers on a variety of operating systems.This unit provides sufficient details of two most useful servlet engines i.e. JRUN and ASP. JRun is a web server extension that allows your web server to be 100% servlet API-compatible and to run Java servlets. JRun is a combination of native code to interface directly to your web server in the fastest way possible, as well as a collection of Java classes that provides the interface layer between your server and the servlets that you run. Microsoft's Active Server Pages (ASP) technology is a powerful tool for building Web sites that incorporate dynamic content, like pages built from databases. It will be particularly beneficial to developers already comfortable with Microsoft's Visual Basic. It also discusses Java Server Pages for the ASP Developer. JSP is similar to ASP, the difference being that the code embedded between the <% and %> tags in JSP is exclusively Java.

Common Gateway Interface, or CGI, has been the dominant interface for extending web servers for years. Because CGI support was built into every web server on the market, CGI was a great choice for development tools and applications that could add dynamic capabilities to your web site. Traditionally, CGI languages included C, C++, and Perl, with Perl being the dominant CGI language.

The Java servlet API was introduced as a means of standardizing the use of Java when used through a web server. Servlets are, in fact, a complete replacement for all of today's CGI. Servlets offer many added benefits to the developer including ease of development, automatic garbage collection, fast throughput and response, inter-servlet communications, as well as all of the features inherent in Java.

In the past, the Java Web Server was the only server to fully support the servlet API. The problem is that users already have a large investment in their current server architecture. In addition, the vast array of tools and accessories for today's web server are not available for the Java Web Server. The JRun Servlet Engine, or JRun for short, was developed as an answer to these limitations.

JRun is a web server extension that allows your web server to be 100% servlet API-compatible and to run Java servlets. JRun is a combination of native code to interface directly to your web server in the fastest way possible, as well as a collection of Java classes that provides the interface layer between your server and the servlets that you run. Next, we'll cover the features and capabilities of the JRun system.

2.37 NGWS

You are provided with a runtime environment by NGWS,the NGWS runtime.This runtime manages the execution of code, and it provides services that make programming easier . As long as the compiler that you use supports this runtime, you will benefit from this managed execution environment. Your guess-that the c# compiler supports the NGWS runtime - is correct . However, it is not the only compiler that supports the NGWS runtime;Visual Basic and C++ do so also. The code that these compilers generate for NGWS runtime support is called managed code . The benefits your applications gains from the NGWS runtime are

1. Cross-language integration (through the Common Language Specification).
2. Automatic memory management (garbage collection).
3. Cross-language exception handling (unified unwinding).
4. Enhanced security (including type safety).
5. Versioning support (the end of "DLL hell").
6. Simplified model for component interaction.

For the NGWS runtime to provide all these benefits , the compiler must emit meta-data along with the managed code. The matedata describes the types in your code, and is stored along with your code (in the same PE-portable executable-file).

As you can see from the many cross-language features, the NGWS runtime is mainly about tight integration across multiple different programming languages. This support goes as far as allowing you to derive a C# class from a Visual Basic object (given that certain prerequisites that we'll discuss later are met).

One feature that C# programmers will like is that they don't have to worry about memory management - namely the all-famous memory leaks. The NGWS runtime provides the memory management, and the garbage collector releases the objects or variables when their lifetimes are over - when they are no longer refrenced.

There are even bonuses when deploying a managed application or component . Because managed applications contain metadata, the NGWS runtime can use this information to ensure that your application has the specified versions ofeverything it needs. The net result is that your code is likely to break because some dependency is not met.Another advantage of the metadata approach is that type information resides in the same file where the code resides- no more problems with the Registry.

QUESTIONS WITH ANSWERS

Q.1 Does the .NET Framework only apply to people building Web sites?

Ans. The .NET Framework enables you to create great Web applications. However, it can also help you build the same applications you build today. If you write any Windows software (using ATL/COM, MFC, Microsoftr Visual Basicr, or even standard Microsoftr Win32r), .NET offers many advantages to the way you currently build applications. Of course, if you do develop Web sites, then the .NET Framework has a lot to interest you by starting with ASP.NET.

Q.2 Where can one get the .NET Framework SDK?

Ans: The Beta 1 of the .NET Framework SDK is now available for public download at MSDN Online Downloads. Because of its size, Microsoft offers this beta as a single download (106 MB), or an 11-part download.

Q.3 On what platforms will the .NET Framework run?

Ans: The Beta 1 version will run on Microsoftr Windowsr 2000, Windows 95/98/ME, and Windows NTr 4.0. There is also a version of the .NET Framework called the .NET Compact Framework. It is designed to bring some of the capabilities of the .NET Framework to devices such as cell phones and enhanced televisions. The .NET Compact Framework will run on Windows CE and other embedded operating systems.

Q.4 What programming languages the .NET Framework supports?

Ans: The .NET Framework is language neutral; virtually any language can target the .NET Framework. Currently, you can build .NET programs in a number of languages, including C++, Microsoftr Visual Basic.NET, JScriptr, and C#. A large number of third-party languages will also be available for building .NET Framework applications. These languages include COBOL, Eiffel, Perl, Python, Smalltalk, and others.

Q.5 What is the relationship between the .NET Framework and COM+ Services?

Ans: The .NET Framework gives you full access to COM+ services, while also making it easier to build serviced components. .NET Framework components can be added to a COM+

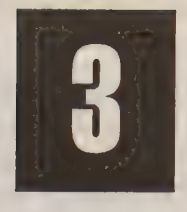
application. There they can take advantage of automatic component services such as transactions, object pooling, queued components, events, and so on.

Q.6. What is the relationship between the .NET Framework and DCOM?

Ans: DCOM is the COM infrastructure for cross-process communication. The .NET Framework supports a number of pluggable channels and formatters for cross-process communication. When making transitions between managed and unmanaged code, the .NET Framework uses the COM infrastructure, specifically, DCOM. All scenarios using COM+ services use managed-to-unmanaged transitions, and thus use DCOM by default. The .NET Framework also supports SOAP, the Simple Object Access Protocol, for cross-process communication where interoperability is critical.

Q.7 Is the .NET Framework just a new name for Windows DNA?

Ans: Windows DNA is an architecture for building tightly-coupled, distributed Web applications. When distributed applications started demanding more loosely-coupled principles, Microsoft evolved the architecture to .NET. The .NET Framework is a part of the .NET architecture.



C# AND OTHER PLATFORMS

3.1 Object Oriented Language Design

The four steps in the order of advance in which computer programming is learnt are as follows:

1 Unstructured programming;
2 Procedural programming;
3 Modular programming;and
4 Object-oriented programming.

3.2 Procedural Languages

Almost any procedure may be depicted as a program in some programming language. In science,particularly in physics , FORTRAN was popularly used for over 40 years. It has fast algorithms as well as millions of lines of codes for different applications. Generally when we develop a new program,we take a known algorithm, and a piece of an already tested and used code and upgrade it with new features. Each statement in such a language tells the computer to do something:get some input, add these numbers, divide by 6,display the output etc.Such a procedural approach is adopted in some other languages such as PASCAL, BASIC etc. A program in a procedural language is a list of instructions.

For small programs, no other organizing principle (often called a paradigm) is needed. The programmer creates the list of instructions, and the computer carries them out.

3.3 Object-Oriented Programming

The structured programming where data is stored separately from procedural code, has some inherent weaknesses. A code written as structured is not modular. Since data elements are accessible from any code,they are liable to be modified without the developer's knowledge.This can result in run-time errors that are difficult to debug.The global impact of a single line of code may be very intricate.Simultaneous management of code and data by the programmer reduces the rate of reuse.

Object-oriented Programming(OOPs)solves these problems. Here, the data and the method are contained in a single unit called an object. The data can be hidden to prevent unauthorized modification. In addition, the object advertises a set of public methods to operate on this data. This concept is called encapsulation. Since implementation details are separated from the interface, the programming logic can be changed without running the risk of breaking the code that calls the object.

OOPs also allows reuse of codes by a process called inheritance. It facilitates easy construction of complex codes from already tested and used codes.This saves time and reduces the risk of errors.

Object oriented languages implement "data abstraction" by using a concept called "classes". We will consider data abstraction in more detail later on since it is the central concept in C++. Data abstraction combines the data and the functions used to manipulate the data so that implementation details are not available to the programmer. Data abstraction makes programs much easier to maintain and upgrade. Programs are thus split into reusable objects. Existing objects can also be extended. By giving programmers immense scope to reuse code, and by virtually forcing programmers to write codes this way, it renders writing new programs a matter of just assembling existing pieces.This is a unique and powerful concept, because something is changed without actually changing its content.The concepts of inheritance and polymorphism make this possible.The existing object stays the same, changes are just layered on it. The programmer is thus easily able to control bugs.The above advantages are common to any object oriented programming language. C++ is equipped with two other advantages over the C language.

C++ is a matured language but still has scope for improvement. In this course, we will explore the possibilities of a new paradigm for Object Oriented Programming (OOPs).

C++accomodates the four main features of OOPs, viz, encapsulation, information hiding, inheritance, and polymorphism.

Object Oriented Programming is a software development technique that has gained popularity because of the potential gains in programmer productivity over conventional software development methodologies which may be attributed to the features of:

1. Encapsulation
2. Information hiding
3. Data encapsulation or abstraction
4. Inheritance

The fundamental idea object-oriented languages is to combine the data and the functions that operate on that data into a single unit called the object.

An object is a self-contained module of data and code which has the data and the procedures related to the object defined within it. So, procedures(called methods in OOPs terminology) are tied to the data in contrast to procedural languages which pass data to procedures. A 'message' identifying a method to execute is passed to the object.

Data abstraction is a significant benefit offered by OOPs. Data is declared local to an object.This declaration can be either private or public. Hiding data from other parts of a program makes it possible to totally change the data structure of an object withou taffecting the operation of the program.

An object's functions, called member functions in C++, provide the only way to access its data. If you want to read a data item in an object, you call a member function in the object. It will read the item and return the value to you. You can't access the data directly. The data is hidden, so it is safe from accidental alteration.Data and its functions are said to be encapsulated into a single entity. Data encapsulation and data hiding are the key terms in the description of object oriented languages.

No function other than the member function of an object can access the data. This simplifies writing, debugging and maintaining the program.

3.4 Encapsulation

What is perhaps most important (and different) about the object- oriented approach to software is that it unifies inside 'objects' two things that have traditionally been separated in programming paradigms:The data and the method.

Encapsulation allows the implementation of 'information hiding'—restricted access to the internal representations of objects..Objects have a 'private' part of hidden internal detail (state data and internal processes), and a 'public' interface which clearly defines the set of possible behaviours of the object (the messages to which it can respond).

```
Object = (private) data + (public) processes
```

This separation of the public interface of an object and its internal implementation allows us to treat the two as separate parts of the programming process. An object's private implementation details can be defined without reference to other objects or a particular application context, and then the object may be used in an application that utilizes only the object's public interface. With encapsulation, we can accomplish data hiding. Data hiding is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally.

C++ supports the properties of encapsulation and data hiding by creating user defined types, called classes. Once created, a class can be used as a whole unit. Users are not needed to know how the class works; they just need to know how to use it. Both the public and private parts of an object are defined in an 'abstract data type' which defines the common aspects of all objects in a particular 'class', i.e. objects that we classify as one type. All the operations performed on that data are linked together.This makes a program comparable to real world situations.

A real world object has, a name(identity), some attribites(state) and some functions(behaviour).These elements are all 'encapsulated' into one object—we cannot divorce any one of these from the others. For example, the state of being full is irrevocably linked to the behavior of being filled—we cannot separate the two. If a cup has not been filled then it cannot be full. In an object-oriented software, a similar link exists between the data and the operation performed on it.

3.5 Information Hiding

As mentioned earlier, encapsulation enables us us to implement 'information hiding' (a term first coined by David Parnasin in 1972). By this, we mean that encapsulation hides private elements of an object behind a public interface. This separation between the public and private parts of an encapsulated object provides for concealing the processing details as well as safety from unwanted manipulations.

The state of an object can be altered only by using the methods that are fixed. If we take the coffee cup example, we can say that it has a state of 'fullness' (how much coffee is in the cup), just as the state of fullness of the cup can be changed only by the method of emptying, the state of an object can be changed only by some fixed methods.This protects the data from changing under the influence of an unconnected, inadvertent influence.

3.6 Abstraction and Encapsulation

Abstraction is the representation of all the essential features of an object, which represents its possible states and behaviors. These are 'encapsulated' into an 'abstract data type' which defines how all objects in a class (type) of objects are to be represented and how they behave.

Encapsulation is the practice of including in an object both data and processes hidden from other objects in the system.The internal state of an object is not directly accessible from outside, and cannot be altered by external changes to the application. Similarly, changes to the internal implementation details can be made without affecting the external interface.

3.7 Inheritance

The idea of classes leads to the idea of inheritance.

We use the term class and sub-class very frequently in our lives. For instance, we may say that both the lion and the tiger belong to the class of carnivores , each belonging to a separate sub-class. Regarding inheritance, we are liable to say that each class has some attributes of its own as also that they have some common attributes inherited from the common class.

Similarly, a class in OOPs can be a derived class which has inherited some attributes from a base class besides possessing some attributes of its own.

The relation between classes and objects is different from the relation between a base class and the derived classes. Objects embody all the characteristics of their class and are designs that exit in the memory of the computer.

In procedural programs, if, say, three sections of a program do the same thing, we may combine them in a single function so as to shorten the program. We can also cause the function to perform individual processing particular to it. In OOPs, the same logic is used to implement inheritance of classes. Elements common to many derived classes may be caused to be inherited from a common base class.The derived classes can, besides have individual elements too.

Multiple inheritance means, a derived class inheriting attributes from more than one base class.The inheritance relationship among various classes is referred to as a classification hierarchy.The concept of inheritance allows reuse of types. A new type can be derived from an existing one, declared and used independently. Such types are called derived types.

3.8 Polymorphism

Different objects can be made to respond in different manners to the same message.This process is referred to as polymorphism.The responses are specific to the type of the object. Symbols like + or a function like 'show' may be over-loaded so as to imply different meanings depending on the context.While dealing with dynamic objects, that is, objects that are created and destroyed

during run-time, we cannot predict the number of objects that will receive a particular message. Hence, we have to defer our decision about the meaning of an object until run-time or in other words, we have to resort to what is called dynamic binding. Dynamic binding leads to run-time polymorphism.

3.9 Advantages of OOPs

OOPs allows reuse of classes already tested and successfully used. New programs can thus use these classes maybe with new objects. Objects are easy to create, test and debug since they contain small code blocks.This definitely leads to time-saving.

Different programs can be developed in almost similar lines by ajudicious selection of the class objects.This technique of data abstraction is the essence of OOPs.The need to go for a new language for every situation is eliminated by OOPs.In fact the spawning of too many languages was one factor that motivated the creation of OOPs.Another factor was the over use of the goto statement in structured programs.It was often necessary to use this statement over and over and some structured programs were rendered incomprehensible and difficult to debug because of this.OOPs presents a convenient alternative to this situation.

3.10 C++ and other Object Oriented programming (OOPs) Languages

Simula67- the first object-oriented language appeared in 1986 and was readily hailed. Then came a group of these, all having thier origin in Simula. Simula was developed in Norway as a discrete event simulation language, and was the first language to introduce the key object-oriented concepts of classes and inheritance.

C++ has been derived from C language.Strictly speaking, it is a superset of C: almost every correct statement in C is also a correct statement in C++, although the reverse is not true. the most important elements added to C to create C++ are converged classes, objects, and object-oriented programming. (C++ was originally called "c with classes"). However, c++ has some other new features as well, including an improved approach to I/O and a new way to write comments.

C++ extends into Java. The Java language uses a structure that is almost identical to C++. Infact, the Java designers considered C++ to be too complicated and therefore avoided some of its complicated features and thus Java is without such features of C++ . The Java people make a good case for this, but largely it is a matter of personal taste whether you prefer the Java or the C++. However, if you are familiar with C++ you can easily understand Java. The rudiments of C++ can be mastered in a few hours.

3.11 Evolution of C++ Programming Language

The C language was developed at Bell Laboratories (now AT&T) in the early1970's by a programmer named Dennis Ritchie. It was written originally for programming under an operating system called UNIX,which itself was later rewritten almost entirely. When the language was modified and improved to its present state,C,was chosen to represent the new version.

The C++ was designed and developed by Bjarne Stroustrup in the computer Science Research Center at AT&T Bell Labs.

C++ isa classic third generation procedural language. In order to program in C++ knowledge of C is desirable. C++ gets its name from one of the standard C++ operators. Taken literally, it would mean "one more than C".

C++ is currently one of the most popular third generation languages. It achieved popularity as a basic development platform form for any operating system.

The use of C++ today exceeds the use of all other programming languages put together. It is by no means certain that this situation will persist; many academic organizations for instance prefer java to C++.The riseof 'visual' programming languages like Microsoft's 'Visual BASIC ' has also had an impact on the use of C++. However,this should not be a deterrent to anyone who wants to learn C++ . On the contrary, a good understanding of C++ will make understanding Java a simple exercise, and will render the use of languages like Visual BASIC unnecessary. With practise, you wil lbe able to produce better programs more quickly using C++ than any other system.

C++ implements all of C's features, and adds some more. Most of these enhancements are beyond the scope of this course,but we will use them when they are simpler or more logical than the equivalent features.The most profound enhancement of C++ over C is the addition of 'object oriented' features.

Object-orientation is a technique for handling the complexity of a large program by dividing it into self-contained components (called classes) that often model features in the real world. Object orientation is one of the most important advances in computer science in recent decades.

3.12 Distinctive Features of C++ Language

Strictly speaking,C++ is more of an instruction code than a language.It has its own set of rules or syntax.

It is often described as a"middle-level" language.It permits programs to be written in much the same style as that of modern high-level languages, such as FORTRAN, COBOL, BASIC, PL/I, and Pascal. Where it differs is that it permits very close interaction with the inner working of the computer. It makes possible dealing with the machine at a fairly low level. Even though C++ is concise and terse, it is extremely powerful. When using a procedural language, a programmer writes explicit directions for a computer about the steps it is to perform.The programmer must understand the meaning of the data, the step being performed, and the result. A high level language frees the programmer from having to work with machine instructions and from having to determine the locations of data in main storage.These details are provided by a compiler,which translates a program from source code, in the high-level language, to machine instructions.

3.13 Limitations of Using C++

1.The flexibility provided in C++ can be a problem for an inexperienced programmer, because a mistake in a program will often result in some valid, although unwanted interpretation. Programmers occasionally over use compact forms of statements and create programs that are

hard to read, understand, and test.To be used effectively, C++ requires a more developed programming. C++ does not provide a machine-independent way to specify the precision of data.There are some generally observed practical standards,explained in this book, that meet most needs.

For numerical analysis, however, this deficiency can become a major problem.Other languages ,particularly FORTRAN, are generally preferred over C++ in this regard.C++ has no input or output facilities. Under Unix, these Services are provided through functions available to any C++ program that is not part of the operating system. The Unix services have become a practical standard. A Unix format function library is supplied with most C++ compilers. Unix services are oriented toward unstructured, printable characters in files.

Additional services are needed by programs that respond rapidly to interrupts, process binary data, produce displays, handle high-speed communications, and use large or numerous files of interrelated data. For many applications, additional services are provided by writing part of a program in an assembler language or by linking C++ code to code written in another high-level language.For certain applications, such as financial accounting, engineering data analysis, and high-reliability control systems, other languages-COBOL, APL, or Ada-may prove to be better.

3.14 Basic programming Features of C++ Language

Here are some of the aspects of C++ that will be used for the first programming assignment.

A simple program:

The empty shell of a program looks like

```
main()
{
}
```

A program that actually does something will appear like:

```
#include <iostream.h>
main()
{
  cout << "Hello. \n";
}
```

```
*  cout is the "print to screen" function
*  #include <iostream.h> puts in the function prototype for cout
*  << sends the word "Hello." to cout
*  \n puts in a return
*  {} denote a block of code
*  ; is the end of a statement
```

3.14.1 Commments

There are two ways to denote comments:

```
* A comment can start with /
* and end with
*/ *
Or // denotes commenting out the rest of the line
```

3.14.2 Input & output

There are many ways to get information into and out of a program. For example, a command that writes a line

```
cout << "Hello. \n";
```

or a command that reads a line

```
cin >> num_repeats;
```

Both these commands require that the program read a definition of how the functions work with a line like this

```
#include <iostream.h>
```

at the top of the program.

3.14.3 Variables

Variables must be declared with lines like these:

```
int  account_number;
float interest_rate;
char middle_initial;
char last_name[80];
```

The last line declares an 80 character long string.

Variables declared inside a function are called "local variables" . Local variables are accessible only by commands within that function.Variables declared outside any of the functions are called "global variables". Global variables can be accessed by commands at any point in the program. All variables must be declared.

3.14.4 LOOPs

A lOOPs is created with the command

```
for ( counter=0 ; counter<5 ; counter++ )
  {
    // commands here are repeated 5 times
  }
```

3.14.5 Constants

A constant can be declared with a command like

```
const float PI = 3.1416;
```

3.14.6 Conditional statements

The value of a variable can be tested with the "if" command like this

```
if ( x==1 )
  {
    // execute these statements
  }
```

Note that the test operator is two equals signs in a row. A common mistake is to write an "if" like this

```
if ( x=1 )
  {
    // execute these statements
  }
```

This time there is a single equals sign. The compiler will not indicate any error here, but the condition will be seen as true every time. With == we are asking whether x is equal to one. With =we are asking whether the program was able to successfully assign the value of one to x.

3.14.7 Relational O7perators

The relational operators are

```
== equal != not equal
< less than
> greater than
<= less than or equal to
>= greater than or equal to
```

It is also possible to string operators together with the symbols && denoting a logical "and" and || denoting a logical "or". For example

```
if ( (x==3) && (y<7) )
  {
    // execute this code
  }
```

It is also possible to carry out one action if a test is true and another action if the test is not true with a command like this

```
if ( x==1 )
  {
    // do this if true
  }
 else
  {
    // do this if false
  }
```

This idea can be taken farther with a structure like this

```
if ( test1 )
  {     }
else if ( test2 )
  {     }
else if ( test3 )
  {     }
else
  {     }
```

3.14.8 Repeating

Another way to use a test is to keep repeating a set of commands until some condition is met, like this

```
do
  {
     // repeat until x is not less than one
  } while ( x<1 );
```

In this case it is necessary for the commands inside the LOOPs to be changing the value of x in some way. Otherwise, this will be an infinite LOOPs.

3.15 Java

The most common types of programs written in the Java programming language are applets and applications. If you've surfed the Web, you're probably already familiar with applets. An applet is a program that adheres to certain conventions that allow it to run withina Java-enabled browser. At the beginning of this trail is an applet that displays an animation of the Java technology's mascot,a Duke, waving at you. However, the Java programming language is not just for writing cute,entertaining applets for the Web. The general-purpose, high-level Java programming language is also a powerful software platform.Using the generous API, you can write many types of programs.

An application is a stand alone program that runs directly on the Javaplatform.Aspecial kind of application known as a server serves and supports clients on a network. Examples of servers are Webservers,proxy servers,mailservers, and print servers. Another specialized program is a servlet. A servlet can almost be though tof as an applet that runs on the server side. Java Servlets are a popular choice for building interactive web applications,replacing the use of CGI scripts. Servlets are similar to applets in that they are run time extensions of applications.Instead of working in browsers, servlets run within Java Web servers, configuring or tailoring the server.

How does the API support all these kinds of programs? It does so with packages of software components that provide a wide range of functionality. Every full implementation of the Java platform gives you the following features:

1. **The essentials:** Objects, strings, threads, numbers, input and output, data structures, system properties, date and time, and so on.
2. **Applets:** The set of conventions used by applets.
3. **Networking:** URLs,TCP(TransmissionControl Protocol), UDP (User Datagram Protocol) sockets, and IP (Internet Protocol) addresses.
4. **Internationalization:** Help forwriting programs that can be localized for users world wide. Programs can automatically adapt to specific locales and be displayed in the appropriate language.
5. **Security:** Both low level and high level, including electronic signatures, public and private key management, access control, and certificates.

6. **Software components:** Known asJava Beans,can plug into existing component architectures.
7. **Object serialization:** Allows light weight persistence and communication via Remote Method Invocation (RMI).
8. **Java Database Connectivity (JDBCTM):** Provides uniform access to a wide range of relational databases.

The Java platform also has APIs for 2D and 3Dgraphics, accessibility, servers, collaboration,telephony,speech, animation, andmore. The following figure depicts what is included in the Java 2 SDK.

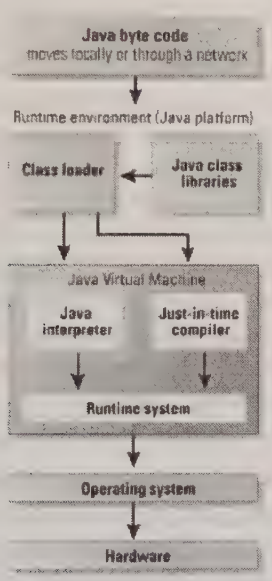


Fig. 3.1

Note: The Java 2 SDK, Standard Edition v. 1.3. The Java 2 Runtime Environment (JRE) consist s of the virtual machine, the Java platform core classes,and supporting files.The Java 2 SDK includes the JRE and development tools such as compilers and debuggers.

3.16 Comparison of C#, C++ and VB.net

The most significant difference between programming with the supported languages is not their language syntax. BothC++and VisualBasic have an established history and significant base of existing code, which were taken into account when they were updated for .NET. C#, on the other hand, starts with a cleaner slate. Several advanced topics - such as the build process and defining name spaces- will be introduced with the corresponding sample code in the following sections, but here are some of the more obvious differences between two the languages.

CaseSensitivity:C++ and C# are both case-sensitive, but VB is not.For a program to be CLS compliant, however, public members cannot differ only in their case. This restriction allows VB (and potentially otherCLS-compliant languages) to both produce and use components created in other, case-sensitive, languages.

Referencing a Library:To use classes,which are located in name spaces,it is first necessary to obtain a reference to the assembly containing the desired name space. All .NET programs use -ataminimum-the System name space,which is found in mscorlib.dll (typically located in the Windows System directory):

```
C++ #using <mscorlib.dll> C# (implicitly loaded)
```

VB (implicitly loaded)

Note that Managed Extensions uses a preprocessor directive (this may change before final release) and the file name of the DLL. C# and VB currently references the System assembly implicitly (this may also change), but for other assemblies it is necessary to use the /reference compile switch. Referenced libraries are generally located in the application directory,or a sub directory of the application. Libraries that are designed for use by many applications- for instance, tools provided by 3rd parties - are located in the assembly cache (currently %system%\assembly) and must follow specific guidelines. Application configuration files can provide additional options. For ASP.NET applications, however, components should be locatedin "\Bin" subdirectory under the starting point for the application's virtual directory.

Importing a Namespace:Classes can be either referenced fully (e.g. System. IO.FileStream,similar to a fully qualified path name)or their namespace can be imported into the program, after which it is not necessary to fully qualify the contained class names.For convenient access to System objects, that namespace must be imported:

C++ using namespace System; C# using System; VB Imports System

Note that both Managed Extensions and C# use a using statement, while VB uses Imports.

ReferencingObject Members: Both VB and C# support the period as as copere solution operator, which allows (for example) the syntax Console.WriteLine when referencing the WriteLine method of the Console object. C++ uses a double colon "::" as a resolution operator:

C++ Console: WriteLine("xxxxx"); C# Console.WriteLine("xxxxx"); VB Console. WriteLine "xxxxx"

DeclaringObjects: In Managed Extensions for C++ and C# (though notinVB), variables must be declared before they can be used. Objects are instantiated using the new keyword. The following are sample declaration / creation statements - declaring and creating an object of type Comp, in namespace Lib, with the name myComp - in each of the three languages:

C++ Lib::Comp* myComp = new Lib::Comp(); C# Lib.Comp myComp = new Lib.Comp(); VB Dim myComp As New Lib.Comp

Program Entry Point: Every executable program has to have an external entry point, where the application begins its execution. The syntax hasn't changed for Managed Extensions, but in C# and VB everything happens in a class:

```
C++   void main() { } C#   class MainApp {
   public static void Main() {
   }
}
```

```
VB    Public Module modmain
      Sub Main()
      End Sub
   End Module
```

Behind the scenes,however,theManagedExtensions compiler encapsulate the entry point in a class.

Defininga Namespace & Class: Each of the three languages supports the creation of custom name spaces as well as classes within those namespaces. All three languages handle this in code, though with slightly different syntax. For instance, note that the "managed" class declaration for C++ and the fact that trailing semicolons are not needed for namespace and class declarations in C#:

```
C++    namespace CompVC {
       __gc class StringComponent {
       public:
          StringComponent() {
          }
       };
   };
C#    namespace CompCS {
       public class StringComponent {
          public StringComponent() {_}
       }
   }
VB    Namespace CompVB
       Public Class StringComponent
          Public Sub New()
          End Sub
       End Class
   End Namespace
```

Overall, C# provides greater expressiveness and is more suited to writing performance-critical code than Java, while sharing Java's elegance and simplicity, which makes both much more appealing than C++.

3.17 XML and Java

the power to Java may be attributed to the universal nature of XML as data and/or documents. Java has become the universal programming language, used for embedded systems in small devices upto large main frame class servers.

Documents have typically two issues that make them difficult to be understood by external programs: proprietary formats and generalized formats. Proprietary formats (such as Microsoft WordDocuments) are difficult to connect to because they are not plain text,but HTML documents, and the immense store of information on the Web is difficult for a program to understand due to the limitations of a vague medium. A Word document, a spreadsheet, or even a web page is difficult to parse even when you know what you are looking for.

Now,something about XML.

XML is a language similar to HTML but more akin to their common ancestor, the SGML or Standard Generalized Markup Language.SGML,is a meta-language, and provides for creating your own markup language. This faculty was exploited by Tim Berners-Lee who created HTML and the Web. XML,unlikeHTML is not a specific language with a common set of tags (elements and attributes) that are understood by several parties (Netscape and Microsoft). XML is like SGML, a meta-language.

This means that XML allows the user to create specific languages for describing the data handled. For example,one can describe his/her inventory one way,and another can describe his/her inventory another way. If they were to interface with each other, it would help if both used the same format, that way the first inventory program can work with the second data and vice-versa. But, agreeing on the same format is often unrealistic and a waste of time,for, one will then, need to run his/her business as does another.

With XML one can describe his/her inventory in a document using his/her own "language".As long as this document is XML compliant, any XML parser can pull it into a computer program very easily and build an inventory tree using the DOM (Document Object Model). Even if another may have a completely different inventory system and a completely different format or XML language,a simple bridge can connect the two DOM's together as and when they choose.

Java allows all the different computers to work together because the Java platform runs on top of all common business computer platforms.

SUMMARY

1. A computer program can be thought of as consisting of a set of tasks. Any task that is too complex to be described wholly would be broken down into a set smaller component tasks, until the individual tasks are sufficiently small, simple and easy to uderstand and analyze.
2. Object-orinted programming accomplishes this by providing techniques for man aging enormous complexity achieving reuse of software components and coupling data with the tasks that manipulate that data.
3. The essence of object-oriented programming is to treat data and the procedures that act upon the data as a single "object"- a self contained entity with an indentity and certain charactistics of its own.
4. C++ fully supports object-oriented programming and includes all the four features of object oriented development- encapsulation ,data hiding, inheritance, and polymorphism.
5. Encapsulation is the combination of state and behavior into a software `object'.
6. "Information hiding" is the division of an object into private and public parts. The public part provides the external interface.the private part and all internal processes) are `hidden' inside the object.
7. An abstract data type defines the attributes (state) and methods(behavior)of all objects belonging to a particular `class'.
8. In C++,we use the `class' to model abstract data types. The keywords`public'and `private' are used to define the class interface and hidden part respectively. The scope resolution operator (doublecolon) is used to link member functions (methods) to the class.

4

C# : INTRODUCTION

This lesson will acquaint you with a few simple C# programs. By going through this chapter, you will :

1. Understand the basic structure of a C# program.
2. Obtain a basic familiarization of what a "Namespace" is.
3. Obtain a basic understanding of what a "Class" is.
4. Learn what a "Main" method does.
5. Learn how to obtain command-line input.
6. Learn about console input/output (I/O).
7. Compiling a program
8. Making project in VS.net

4.1 Introduction to C Sharp

C# is a simple, object oriented and type-safe programming language derived from C and C++. C# (pronounced "C sharp") hails from the family the C and C++, and will readily appear familiar to C and C++ programmers. C# aims to combine the productivity of Visual Basic and the power of C++.

C# is provided as part of Microsoft Visual Studio 7.0. In addition to C#, Visual Studio supports Visual Basic, Visual C++, and the scripting languages VBScript and JScript. All these languages offer access to the Microsoft .NET platform, which includes a common execution engine and a rich class library. The Microsoft .NET platform defines a "Common Language Subset" (CLS), that ensures seamless interoperability between CLS-compliant languages and class libraries. Thus C# developers,have complete access to the rich class libraries used by Visual Basic ,Visual C++ etc. C# as such does not include a class library.

The essential features of the language are described in this chapter. The latter chapters describe the rules and exceptions in detail so as to provide the reader with an introduction to the language that will facilitate the writing of early programs and the comprehension of later chapters.

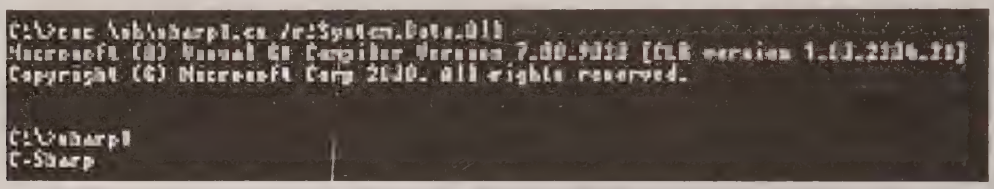
4.2 Main: Entry Point

In a C# Program, classes are included for carrrying out some functions. It is mandatory for every program to get the Main() function (rather method) compiled . This is the entry point. However, it is not necessary to keep this part at the beginning of the program.

Listing 4.1 Minimal C# Program

```
//sharp1.cs
class xyz1 {
   static void Main()
  {

  }
}
```

This program will get compiled easily without any error but will acheive nothing. The compilation process generates an .exe file that is executable and copies it to C:\ directly.

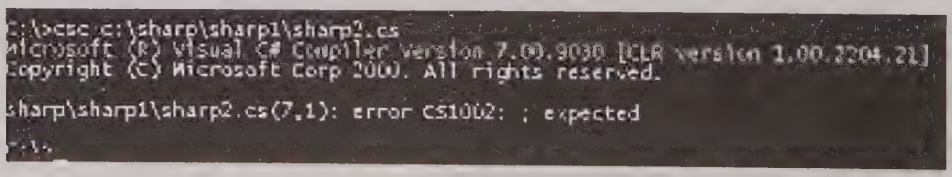


Let us make the program functional by using Writeline(). The pair of brackets () suggest that this is a method. We shall learn about methods (functions) in chapter 7. The Command WriteLine is for outputting any message, prompt or results of computation on the screen.

Listing 4.2

```
//sharp2.cs
class xyz2 {
  static void Main()
  {
    WriteLine("C-Sharp")
  }
}
```

The method WriteLine here is intended to print the phrase "C-Sharp".But when this program is compiled, it will announce the error message:


Microsoft (R) Visual C# Compiler version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
sharp\sharp1\sharp2.cs(7,1): error CS1002: ; expected

The error message tells us that there is an error at line number 7 at column number 1. In fact the error is at the end of the line number 6: of not putting up a ;. In C#, it is necessary to put a semicolon after every method. So put a semicolon at the end of line 6.

Listing 4.3

```
//sharp3.cs
class xyz3 {
  static void Main()
  {
    WriteLine("C-Sharp");
  }
}
```

Run the above program and you will find another error message which is as follows:

```
C:\>csc C:\sharp\sharp1\sharp3.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\sharp\sharp1\sharp3.cs(6,1): error CS0103: The name 'WriteLine' does not exist
in the class or namespace 'xyz3'

C:\>_
```

Actually there is no function WriteLine exclusively but its actual code is `System.Console.WriteLine()`. As such, the program `sharp3.cs` can be modified as follows:

Listing 4.4

```
//sharp4
class xyz4 {
  static void Main()
  {
    System.Console.WriteLine("cSharp");
  }
}
```

This program, actually, is directing to print the message at the console of the system i.e. your PC as shown below:

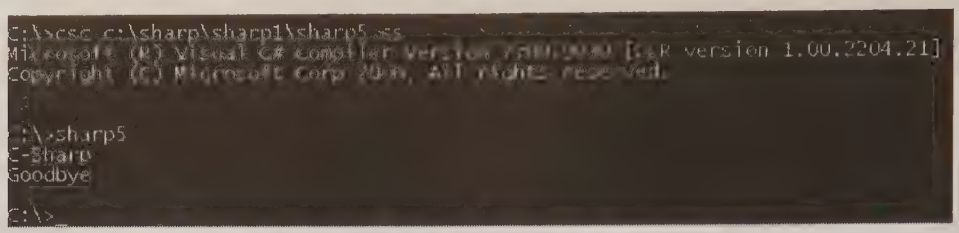
```
C:\>csc c:\sharp\sharp1\sharp4.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>sharp4
cSharp
```

Let us see whether WriteLine prints a new line each time this statement is included in the program:

Listing 4.5

```
//sharp5.cs
class xyz5 {
  static void Main()
  {
    System.Console.WriteLine("C-Sharp");
    System.Console.WriteLine("Goodbye");
  }
}
```

The output of this program shall be displayed on the console as follows:



Every time a function is called, the message is displayed on a newline. Program `sharp6.cs` is an attempt to show how the functions are called and created. This program calls a function alpha().

4.3 First C# Program

The following console program is the C# version of the traditional "Hello World!" program. Here we have deviated a little in giving the message on console as "Hello Universe!" instead of " Hello World!. The program displays the string :Hello Universe!.

Listing 4.6

```
// Printing  "Hello Universe!" program in C#: Hello1.cs
class Hello {
   static void Main()
   {
      System.Console.WriteLine "Hello Universe!");
   }
}
```

The following points are to be noted with regared to this program:

1. Comments
2. The Main Method
3. Input and Output
4. Compilation and Execution
5. Comments

The first line contains the comment:
// Printing "Hello Universe!" program in C#: Hello1.cs

The characters // convert the rest of the line to a comment. You can also comment a block of text by placing it between the characters /* and */, for example:

/* Printing "Hello Universe!" program in C#: Hello1.cs*/

This program displays the string "Hello Universe!" on the screen. If you are conversant with C and C++ programming you may recall that the first method is used in C++ exclussively for writng comments and the second one is used in both C and C++.

4.4 The Main Method

The C# program must contain a Main method, in which control starts and ends. The Main method is where you create objects and execute other methods.

The Main method is a static method that resides inside a class or a struct. In the "Hello Universe!" example, it resides inside the Hello class. There are three ways to declare the Main method:

It can return void: as shown below:

```
static void Main() {
  ....
}
It can also return an int:
static int Main() {
   ...
   return 0;
}
```

It can also take arguments:

```
static int Main(string[] args) {
   ...
   return 0;
}
```

The parameter of the Main method is a string array that represents the command-line arguments used to invoke the program.But unlike in C++, this array does not include the name of the executable (exe) file.

4.5 Another C# Program

The "hello, world" program can be written as follows:

Listing 4.7

```
using System;
class Hello {
     static void Main()
     {
          Console.WriteLine("hello, world");
     }
}
```

The source code for a C# program is typically stored in one or more text files with a file extension of .cs, as in hello.cs. Using the command-line compiler provided with Visual Studio, such a program can be compiled with the command line directive.

csc hello.cs

which produces an executable program named hello.exe. The output of the program is:

hello, world

A close examination of this program will reveal the following facets:

1. The using System; directly references a namespace called System that is provided by the Microsoft .NET Framework class library. This namespace contains the Console class referred to in the Main method. Namespaces provide a hierarchical means of organizing the elements of a class library. A "using" directive enables unqualified use of the types that are members of the namespace. The "hello, world" program uses Console.WriteLine as a shorthand for System.Console.WriteLine.
2. The Main method is a member of the class Hello. It has the static modifier, and thus is a method on the class Hello rather than an instance of this class.
3. The main entry point for a program-the method that is called to begin execution is always a static method named Main.
4. The "hello, world" output is produced through the use of a class library. The language does not itself provide a class library. Instead, it uses a common class library that is also used by languages such as Visual Basic and Visual C++.

The ensuing points with reference to the "hello, world" program will be of interest to C and C++ developers.

1. The program does not use a global method for Main. Methods and variables are not supported at the global level; such elements are always contained within type declarations (e.g., class and struct declarations).
2. The program does not use either "::" or "->" operators. The "::" is not an operator at all, and the "->" operator is used in only a small fraction of programs. The separator "." is used in compound names such as Console.WriteLine.
3. The program does not contain forward declarations. Forward declarations are never needed, as declaration order is not significant.
4. The program does not use #include to import program text. Dependencies among programs are handled symbolically rather than textually. This system eliminates barriers between programs written in different languages. For example, the Console class could be written in another language.

4.6 Program Output

C# programs generally use the input/output services provided by the run-time library of the .NET Framework. The statement:

```
System.Console.WriteLine("Hello Universe!");
```

uses the WriteLine method, one of the output methods of the Console class in the run-time library. It displays its string parameter on the standard output stream followed by a new line. Other Console methods are used for different input and output operations. If you include the following statement at the beginning of the program:

```
using System;
```

you can directly use the System classes and methods without fully qualifying them. For example:

```
Console.WriteLine("Hello Universe!");
```

In this case the complete program would be:

Listing 4.8

```
using System;
class Hello {
    static void Main()
    {
       Console.WriteLine("Hello Universe");
```

The source code for a C# program is typically stored in one or more text files with a file extension of .cs, as in hello.cs. Using the command-line compiler provided with Visual Studio, such a program can be compiled with the command line directive,

```
csc hello.cs
```

which produces an executable program named hello.exe. The output of the program is:

Hello Universe!

A close examination of this program will provide the following guidelines:

1. The main entry point for a program-the method that is called to begin execution-is always a static method named Main.
2. The Main method is a member of the class Hello. It has the static modifier, and so it is a method on the class Hello rather than on instances of this class.
3. The using System; directive references a namespace called System that is provided by the Microsoft .NET Framework class library. This namespace contains the Console class referred to in the Main method. Namespaces provide a hierarchical means of organizing the elements of a class library. A "using" directive enables unqualified use of the types that are members of the namespace. The "hello, world" program uses Console.WriteLine as a shorthand for System.Console.WriteLine.
4. The "hello, world" output is produced through the use of a class library. The language does not itself provide a class library. Instead, it uses a common class library that is also used by languages such as Visual Basic and Visual C++.

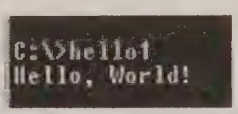
The program does not use #include to import program text. Dependencies among programs are handled symbolically rather than textually. This system eliminates barriers between programs written in different languages. For example, the Console class could be written in another language.

The following examples show different ways of writing the C# *Hello World* program.

Listing 4.9

```
// HelloWorld\Hello1.cs
public class Hello1
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, World!");
    }
}
```

Output



Code Discussion

Every Main method must be contained inside a class (Hello1 in this case). The Main method is the entry point of your program, where the program control starts and ends. It is declared inside a class or struct. It must be static. It can either be void or return an int. The Main method is where you create objects and invoke other methods. The Main method can be written without parameters or with parameters. The latter form allows your program to read command-line arguments.

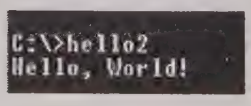
The System.Console class contains a WriteLine method that can be used to display a string to the console.

To avoid fully qualifying classes throughout a program, you can use the using directive as shown below:

Listing 4.10

```
// Hello2.cs
using System;
public class Hello2
{
    public static void Main()
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Output

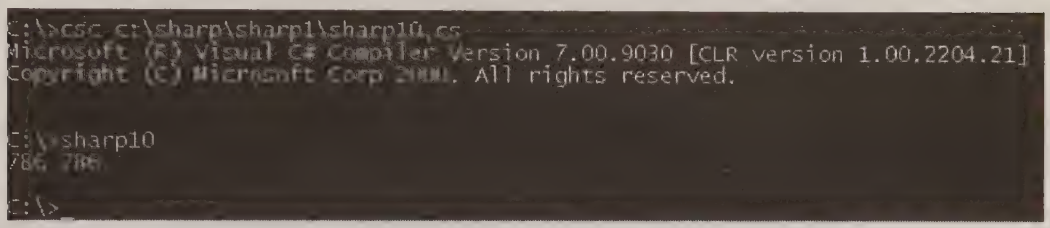


4.7 Printing and Formatting Output

Listing 4.11

```
//sharp10.cs
class xyz10 {
   static void Main()
  {
     System.Console.WriteLine("786 {0}",786);
  }
}
```

The output of this program shall be displayed on console as:


C:\>csc c:\sharp\sharp1\sharp10.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>sharp10
786 786
C:\>

To understand output formatting study the line

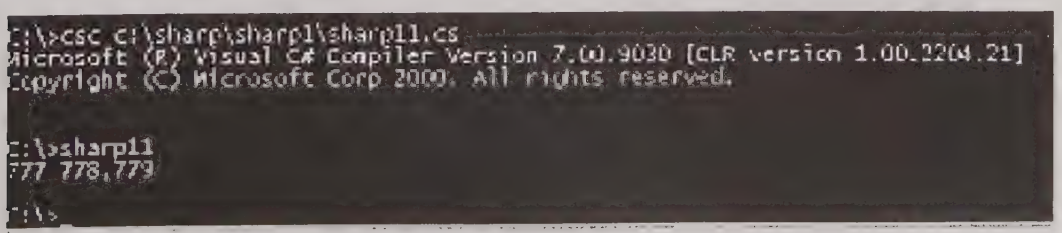
```
System.Console.WriteLine("786 {0}",786);
```

carefully. Whatever is written inside the " " is printed exactly in the output i.e. the first 786. But the 0 inside the {0} tells the computer to print the value of one more item given after the comma , i.e. 786. If you want one more item to be printed then see the program below. Inside the double quotes, the expressions {0}, {1} separated by a comma tells the computer to print two more values given after the second comma.

Listing 4.12

```
//sharp11.cs
class xyz11 {
   static void Main()
  {
      System.Console.WriteLine("777 {0},{1}",778,779);
  }
}
```

The output of this program shall be displayed on console as:


C:\>csc c:\sharp\sharp1\sharp11.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>sharp11
777 778,779
C:\>

In this output the {0} is replaced by 778 and {1} by 779. So it is obvious that {0} means directing the computer to print the first value and {1} means the second value. C# treats 0 to be the first digit and 1 to be second digit.

Listing 4.13 : Using another output Method

```
//input2.cs
using System;
class InputOutput {
     public static void Main()
     {
          Console.Write("Please enter your name: ");
          string strName = Console.ReadLine();
          Console.WriteLine("Hello {0}",strName);
     }
}
```

Line 9 contains a Console.WriteLine statement that uses a formatted string. The format string in this example is

" Hello {0}"

The {0} is replaced for the first variable following the format string in the argument list of the WriteLine method. You can format up to three variables using this technique:

Console.WriteLine("Hello {0} {1}, from {2}", * strFirstname, strLastname, strCity);

Of course, you need not necessarily supply string variables. You can supply any type.

The output of this program shall be:

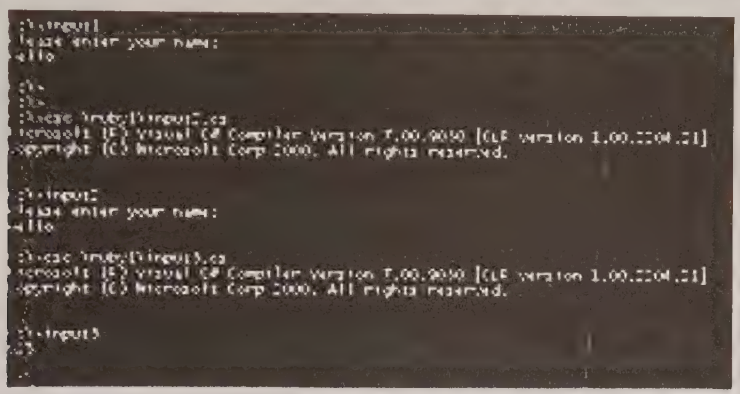
```
C:\ input2
Please enter your name:
Hello
```

4.8 Input from Console

Listing 4.14 : Reading Input from Console

```
//input1.cs
1: using System;
2:
3: class InputOutput
4: {
5: public static void Main()
6: {
7: Console.Write("Please enter your name: ");
8:  string strName = Console.ReadLine();
9: Console.WriteLine("Hello " + strName);
10:      }
11: }
```

Line 7 uses a new method of the Console object for presenting textual information to the user: the Write method . It is different from the WriteLine method only in that it does not add a line break to the output. This approach was used so that the user can enter the name on the same line as the question. After the user enters his name (and presses the Enter key), the input is read into a string variable using the ReadLine method. The name string is concatenated with the "Hello " constant string and presented to the user with the already familiar WriteLine method.
The output of this program will be:



4.9 Compilation and Execution

You can compile the "Hello Universe!" program either by Creating a C# Client EXE Project in the Visual Studio IDE, or by using the command line.

To compile the program from the command line:

Create the source file using any text editor and save it using a name such as Hello.cs. C# source code files use the extension .cs. To invoke the compiler, enter the command:

```
csc Hello.cs
```

If your program does not contain any compilation errors, a Hello.exe file will be created.

To run the program, enter the command:

Hello

The program is compiled and shown below. It is be noted here that the text file name of the program is hello1.cs and this text file resides in directory C:\aw on my computer.

4.10 Namespace Declaration

A Simple Welcome Program : Welc1.cs

Listing 4.15

```
// Namespace Declaration Below
using System;
// Program starting class
class Welcome{
```

```
// Main begins program execution.
public static void Main()
{
   Console.WriteLine("Welcome !");  // Write to console
} }
```

The output of this program is shown below:

```
C:\>csc \welc1.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>welc1
Welcome !

C:\>_
```

The program in listing 4.12 has 4 primary elements, a namespace declaration, a class, a "Main" method, and a program statement.

The namespace declaration indicates that you are referencing the "System" namespace, otherwise you can device your own Namespace. Namespaces contain groups of code that can be called upon by C# programs. With the "using System;" declaration, you are telling your program that it can reference the code in the "System" namespace without pre-pending the word "System" to every reference. We will discuss this in more detail in a later lesson, dedicated specifically to namespaces.

The matter (in curly braces) after class declaration, "class Welcome", contains the data and method definitions that your program uses to execute. It is one of a few different types of elements your program can use to describe objects, such as interfaces and structures, which will be discussed in more detail in a later lesson. This particular class has no data, but it does have a method. This method defines the behavior of this class (or what it is capable of doing).

The method within the Welcome class tells what this class will do when executed. The method name, "Main", is reserved for the starting point of a program. Before the word "Main" is a "static" modifier. The "static" modifier explains that this method works in this specific class only, rather than an instance of the class. This is necessary, because when a program begins, no object instances exist.Classes, objects, and instances will be covered in more detail in a later lesson. Every method must have a return type. In this case it is "void", which means that "Main" does not return a value. (Please note that the return value is not the same as the output of the program).Every method also has a parameter list following it's name with zero or more parameters between parenthesis. For simplicity, we did not add parameters to "Main". Later in this lesson we'll show what type of parameter the "Main" method can have.

The "Main" method specifies it's behavior with the "Console.WriteLine(...)" statement. "Console" is a class in the "System" namespace. "WriteLine(...)" is a method in the "Console" class. We use the ".", dot, operator to separate subordinate program elements. It should be interesting to note that we could also write this statement as "System.Console.WriteLine(...)". This follows the pattern "namespace.class.method" as a fully qualified statement. Had we left out the

"using System" declaration at the top of the program, it would have been mandatory for us to use the fully qualified form "System.Console.WriteLine(...)". This statement is what causes the string, "Welcome !" to print on the console screen.

Observe the first line marked with "//". These are single line comments, which are totally ignored by the compiler. Comments are put up for the various programmers and users to understand the purpose of the program or the program segment. If you wish to span multiple lines with a comment, begin with "/*" and end with "*/". Everything in between is part of the comment. You may place a single line comment within a multi-line comment. However, you can't put multi-line comments within a multi-line comment. Comments are not considered when your program is compiled. They are there to document what your program does in plain English.

All statements end with a ";", semi-colon. Classes and methods begin with "{", left curly brace, and end with a "}", right curly brace. Any statements within and including "{" and "}" define a block. Blocks define scope (or lifetime and visibility) of program elements, which will be discussed in a later lesson.

4.11 Getting Command-Line Input

Many programs are written to accept command-line input. Collection of command-line input occurs in the "Main" method. The following program accepts a name from the command line and writes it to the console.

Listing 4.16 Getting Command-Line Input

```
//welc2.cs using System;
// Program start class class Welcome {
    // Main begins program execution.
    public static void Main(string[] args) {
        // Write to console
       Console.WriteLine("Hello, {0}!", args[0]);
        Console.WriteLine("Welcome to C# !");
    } }
```

Remember to add your name to the command-line, i.e. "Welc2 V.K. Jain". If you don't, your program will crash. It will be shown to you in a subsequent lesson how to detect and avoid such error conditions. The output of this program will be as shown below:

```
C:\>welc2 V.K. Jain
Hello, V.K.!
Welcome to C# !
```

In Listing 4.13, you'll notice an entry in the "Main" method's parameter list. The parameter name is "args". It's what you use to refer to the parameter later in your program. The "string[]" expression defines the Type of parameter that "args" is. The "string" Type holds characters. These characters could form a single word, or multiple words. The "[]", square brackets denote an Array, which is like a list. Therefore, the Type of the "args" parameter, is a list of words from the command-line.

You'll also notice an additional "Console.WriteLine(...)" statement within the "Main" method. The argument list within this statement is different than before. It has a formatted string with a "{0}" parameter embedded in it. The first parameter in a formatted string begins at number 0, the second is 1, and so on. The "{0}" parameter means that the next argument following the end quote will determine what goes in that position. Now we will take up the next argument following the end quote.

This is the "args[0]" argument, which refers to the first string in the "args" array. The first element of an Array is number 0, the second is number 1, and so on. For example, if I wrote "Welc2 V.K.JAIN" on the command-line, the value of "args[0]" would be "V.K.JAIN".But if you write V.K. JAIN i.e the surname written after a gap, computer will ignore the word JAIN in output.

Now we'll get back to the embedded "{0}" parameter in the formatted string. Since "args[0]" is the first argument, after the formatted string, of the "Console.WriteLine()" statement, it's value will be placed into the first embedded parameter of the formatted string. When this command is executed, the value of "args[0]", which is "VK" will replace "{0}" in the formatted string.If you have saved the file by name welc2.cs upon execution of the command-line with "Welc2 V.K. JAIN", the output will be as follows:

```
C:\>welc2 V.K. Jain
Hello, V.K.!
Welcome to C# !
```

4.12 Interactive Input / Output

Another way to provide input to a program is via the console.

Listing 4.14 shows how to obtain interactive input from the user.

Listing 4.17 Getting Interactive Input

```
// welc3.cs using System;
// Program start class class WelcomeName {
    // Main begins program execution.

      public static void Main() {

    // Write to console/get input
        Console.Write("Let me know your Good name?: ");
        Console.Write("Hello, {0}! ", Console.ReadLine());
        Console.WriteLine("Welcome to C# !");
    } }
```

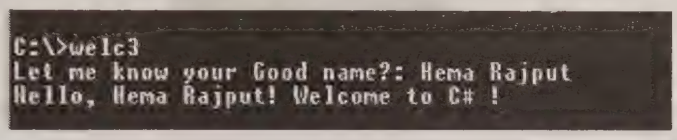
```
C:\>welc3
Let me know your Good name?: Hema Rajput
Hello, Hema Rajput! Welcome to C# !
```

This time, the "Main" method doesn't have any parameters. However, there are now three statements and the first two are different from the third. They are "Console.Write(...)" instead of "Console.WriteLine(...)". The difference is that the "Console.Write(...)" statement writes to the console and stops on the same line, but the "Console.WriteLine(...)" goes to the next line after writing to the console.

The first statement simply writes "What is your name?: " to the console.

The second statement doesn't write anything until it's arguments are properly evaluated. The first argument after the formatted string is "Console.ReadLine()". This causes the program to wait for the user input at the console, followed by a Return or Enter. The return value from this method replaces the "{0}" parameter of the formatted string and is written to the console.

The last statement writes to the console as described earlier. Now if you saved the file with name welc3.cs, upon execution of the command-line with "welc3", the output will be as follows:



By now you know the basic structure of a C# program. You are familiar with namespaces and classes. You know the "Main" method is your entry point to start a C# program. You've also learned how to capture command-line input and perform interactive I/O.

4.13 Command Line Outputs

If you need access to the command line parameters passed in to your application, all you need to do is to simply change the signature of the Main method to include them as shown in Listing 4.18. This example counts and displays the command line arguments.

Listing 4.18

```
//Hello3.cs
using System;

public class Hello3
{
  public static void Main(string[] args)
  {
    Console.WriteLine("Hello, World!");
Console.WriteLine("You entered the following {0} command line arguments:",
args.Length );
    for (int i=0; i < args.Length; i++)
    {
       Console.WriteLine("{0}", args[i]);
    }
  }
}
```

Sample Run
Run the program using some arguments like this: Hello3 qwerty
The output will be:

```
C:\>hello3 qwerty
Hello, World!
You entered the following 1 command line arguments:
qwerty
```

If you enter hello3 X Y Z at command line then output will be:

```
C:\>hello1
Hello, World!
```

Listing 4.19
To return a return code, change the signature of the Main method as shown below:

```
//Hello4.cs
using System;
public class Hello4
{
  public static int Main(string[] args)
  {
    Console.WriteLine("Hello, World!");
    return 0;
  }
}
```

Output
Hello, World!

4.14 Main with return type void and int

The Main method can be of the type void:

```
static void Main() { } //
```

It can also return an int:

```
static int Main() {
   return 0;
```

Example
In this example, the program contains two classes, Factorial and MainClass. The Main method, which resides in the MainClass class, is used to read a number from the keyboard, invoke the Fac() method from the Factorial class, and calculate and display the factorial of the input number.

4.15 Common Language Platform : (CLR)

C# is the new language used for developing applications for Microsoft's .NET platform. This document will go through the basics of this language and explain how you can use it to write efficient platform-neutral applications. The .NET platform is designed to be language neutral, indeed, all .NET code is run under the Common Language Runtime (CLR). C# is just one of the languages that can be used to write classes for the CLR, but so far it is the only language that was written from the outset with .NET in mind. This makes C# as the language of choice for .NET development.

4.16 Two ways of getting output

Notice that there is a newline after each call to `WriteLine()`. There are two ways to get the string and value on the same line, both are shown in the following code:

```
public static void Main()
{
   int i = 1;
   double f = 2.0;
   Console.Write("Integer ");
   Console.WriteLine(i);
   Console.WriteLine("Floating point {0}", f);
}
```

In the first case, we have used the Write() method instead of the WriteLine() method to write the literal string. In the second case, we used another overload of WriteLine() that takes a format string and a variable number of parameters. The format string has placeholders indexed from 0, each one identified by braces ({}). The runtime library will replace the placeholders with the actual values in the variables. The index of the placeholder reflects the order of the parameters to WriteLine(). so the following will print out the integer first, and then the floating point number:

```
Console.WriteLine("Integer {1} Floating point {0}", f, i);
```

4.17 Output Formatting Options

What about formatting options? The common class library comes with classes to allow you to format numbers, which we shall discuss later on. However, Write() and WriteLine() does support column formatting. The syntax is shown in the following example:

```
Console.WriteLine("{0,-10}{1,-3}", "Name","Age"); Console.WriteLine("-
-------"); Console.WriteLine("{0,-10}{1,3}", "Ritik", ritiksAge);
Console.WriteLine("{0,-10}{1,3}", "Jyotsna", jyotsnasAge);
```

Here, the second number in the braces specifies the width of the column and the justification - 10 means that the Name column is 10 characters wide and items are left justified, a value of 10 will make the items right justified. The output from the lines will thus be:

Name Age ----------Ritik 36 Jyotsna 8

4.18 Making Project in Visual C#.Net

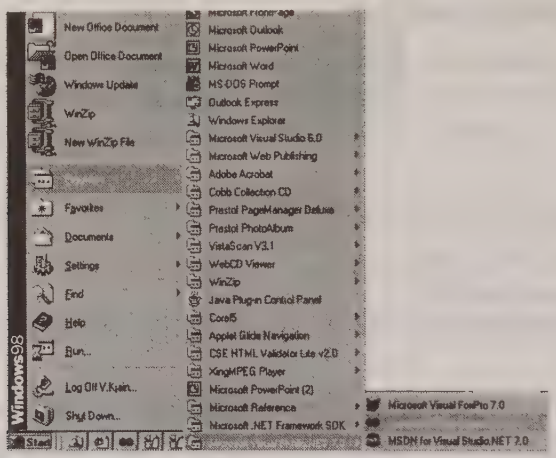
First Visual C# Project

Tools Used : Visual C#.NET

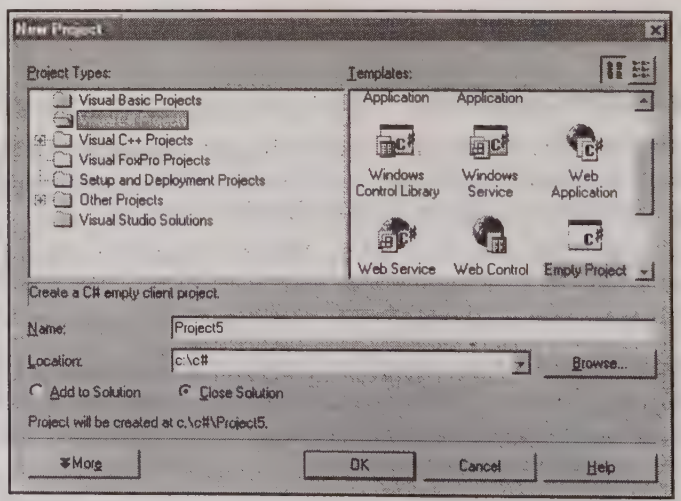
This tutorial guides you towards your first C# Empty project step by step.

Step 1: Creating an Empty Project

1. *Select* File > Program > Microsoft Visual Studio.Net 7.0 > Microsoft Vis0ual Studio 7.0 as shown below:

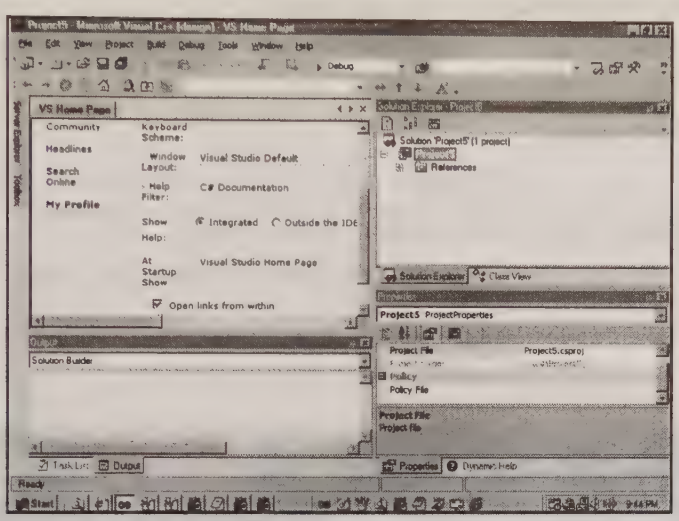


Select File-> New-> Project-> Visual C# Projects -> Empty Project. *Select* your project name and appropriate directory using the Browse button which is as follows:

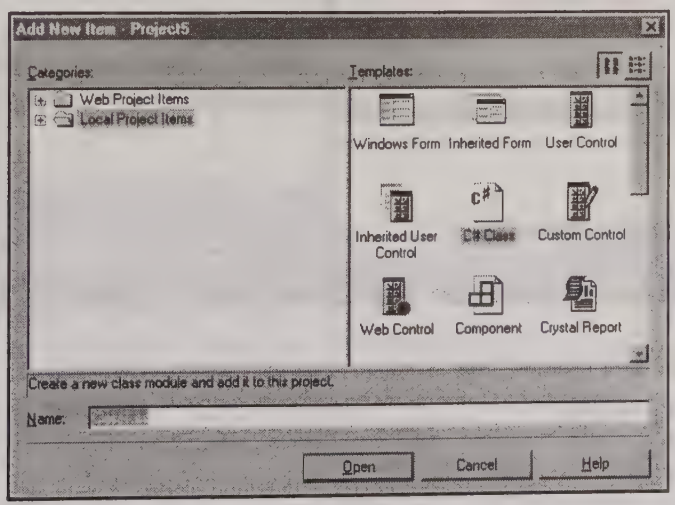


Step 2: Adding an Empty Class

Right-click on the project in Solution Explorer and *click* Add > Add Class

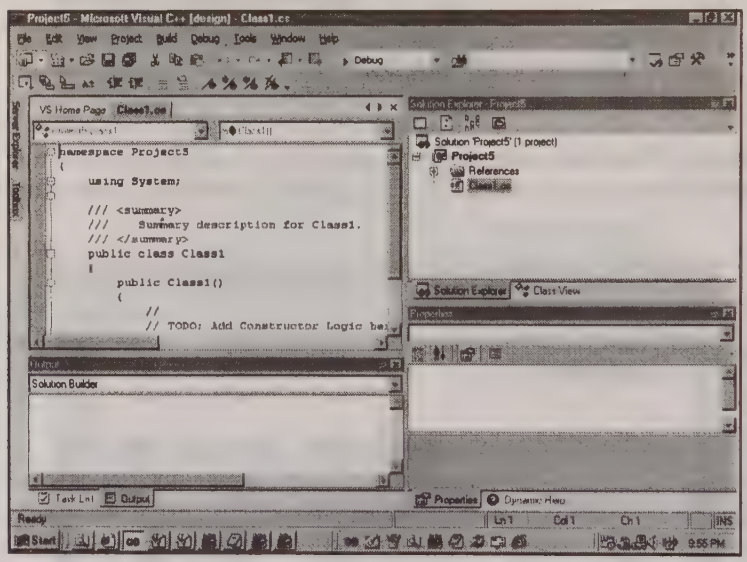


You will see this dialog. There are various kind of classes you can add to your project. Here let us add a C# Class to our project. For this,*write* the name of your class and *click* Open.



Now you have added a class to your project. Visual Studio creates a skeleton of the class for you. It looks like the one mentioned here:

```
namespace DownloadWeb {
   using System;
   /// <summary>
   /// Summary description for McClass.
   /// </summary>
   public class McClass
   {
      public McClass()
      {
         //
         // TODO: Add Constructor Logic here
         //
       }
     }
}
```



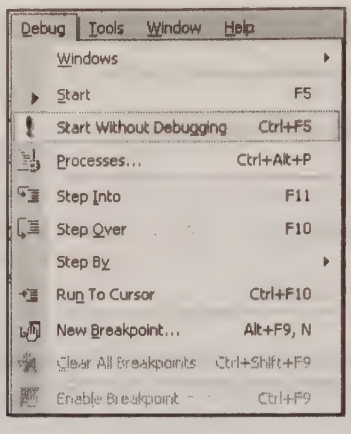
Step 3: *Add* Functionality to your Class

```
namespace DownloadWeb {
   using System;
   /// <summary>
   /// Summary description for McClass.
   /// </summary>
   public class CalCulator
   {
   // Square function
```

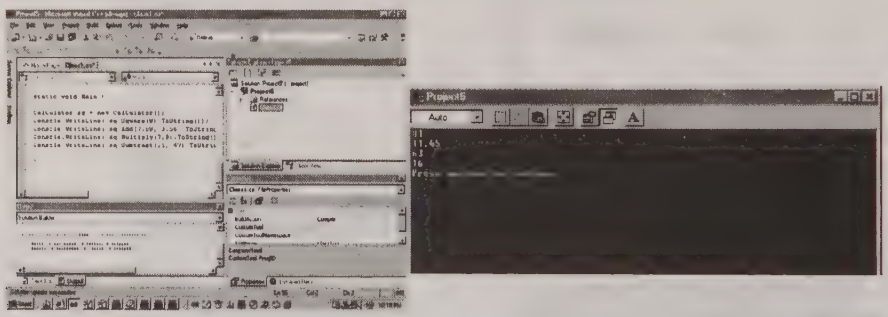
```
        public int Square(int num)
        {
            return num*num;
        }
     // Add two integers and returns the sum
        public int Add(int num1, int num2 )
        {
        return num1 + num2;
        }
    // Add two integers and returns the sum
        public double Add(double num1, double num2 )
        {
             return num1 + num2;
        }
    // Multiply two integers and retuns the result
        public int Multiply(int num1, int num2 )
        {
            return num1 * num2;
         }
    // Subtracts small number from big number
        public int Subtract(int num1, int num2 )
        {
           if ( num1 > num2 )
           {
             return num1 - num2;
           }
             return num2 - num1;
         }
      }

// Call mcClass from Main
public class Class1 {
  static void Main()
  {
   CalCulator sq = new CalCulator();
   Console.WriteLine( sq.Square(9).ToString());
   Console.WriteLine( sq.Add(7.89, 3.56).ToString());
   Console.WriteLine( sq.Multiply(7,9).ToString());
   Console.WriteLine( sq.Subtract(31, 47).ToString());
   }
 }
}
```

Step 4: Building and Running the Project

Choose Debug > Start without Debugging and *click* as shown here:



After clicking, the program will start building up and will report if there are any errors, otherwise, you shall get the following message on the Solution Builder window and output on DOS screen in two different figures.This is shown as follows:



4.19 General Structure of a C# Program

C# programs can consist of one or more files. Each file can contain one or more namespaces. A namespace can contain types such as classes, structs, interfaces, enumerations, and delegates, in addition to other namespaces. The following is the skeleton of a C# program that contains all of these elements.

```
// A skeleton of a C# program
using System; namespace MyNamespace1 {
   class MyClass1
   {
   }
   struct MyStruct
   {
```

```
    }
    interface IMyInterface
    {
    }
    delegate int MyDelegate();
    enum MyEnum
    {
    }
    namespace MyNamespace2
    {
    }
    class MyClass2
    {
       public static void Main(string[] args)
       {
       }
    }
}
```

4.19.1 Program Startup

Program startup occurs when the execution environment calls a designated method, which is referred to as the program's entry point. This entry point method is always named Main, and can have one of the following signatures:

```
static void Main() {_}
static void Main(string[] args) {_}
static int Main() {_}
static int Main(string[] args) {_}
```

As shown, the entry point may optionally return an int value. This return value is used in program termination. A program can be made up of multiple classes or structs, two or more of which contain a method called Main whose definition qualifies it to be used as a program entry point. In such cases, one of these Main methods must be chosen as the entry point so that program startup can occur. This choice of an entry point is beyond the scope of this specification-no mechanism for specifying or determining an entry point is provided.

4.19.2 Program Termination

Program termination returns control to the execution environment. If the return type of the program's entry point method is int, the value returned serves as the program's termination status code. The purpose of this code is to allow communication of success or failure to the execution environment.

4.19.3 Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces (9), which can contain type declarations and nested namespace declarations. Type declarations (9.5) are used to define classes (10), structs (11), interfaces (13), enums 14), and delegates (15). The kinds of members permitted in a type declaration depends on the form of the type declaration. For instance, class declarations can contain declarations for instance constructors (10.10), destructors 10.12), static constructors 10.11), constants 10.3), fields (10.4), methods 10.5), properties 10.6), events (10.7), indexers (10.8), operators 10.9), and nested types.

A declaration defines a name in the declaration space to which the declaration belongs. Except for overloaded constructor, method, indexer, and operator names, it is an error to have two or more declarations that introduce members with the same name in a declaration space. It is never possible for a declaration space to contain different kinds of members with the same name. For example, a declaration space can never contain a field and a method by the same name.

The declaration space of a namespace is "open ended", and two namespace declarations with the same fully qualified name contribute to the same declaration space. For example:

```
namespace Megacorp.Data {
     class Customer
     {
          -
     }
} namespace Megacorp.Data {
     class Order
     {
          -
     }
}
```

The two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names Megacorp.Data.Customer and Megacorp.Data.Order. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a class with the same name.

4.20 Arguments in Main method

The Main method can use arguments, in which case, it takes one of the following forms:

```
static int Main(string[] args)
static void Main(string[] args)
```

The parameter of the Main method is a string array that represents the command-line arguments. Usually you check for the existence of the arguments by testing the Length property, for example:

```
if (args.Length == 0) {
   Console.WriteLine("Please enter a numeric argument.");
   return 1;
}
```

You can also convert the string arguments to numeric types by using the Parse() method. For example, the following statement converts the string to a long number by using the Parse() method on the Int64 class (for more information on System classes, see the Reference section of the .NET SDK):

```
long num = Int64.Parse(args[0]);
```

It is also possible to use the C# type long, which aliases Int64:

```
long num = long.Parse(args[0]);
```

Table 4.1 lists different command line options categorywise.

Table 4.1: Categorywise Command line options

Option	Purpose
Optimization /optimize	Enable/disable optimizations.
Output Files /doc	Process documentation comments to an XML file. /out
	Specify output file. /nooutput
	Compile but do not create an output file.
/target	Specify the format of the output file using one of four options: * /target:exe * /target:library * /target:module * /target:winexe.
.NET Assemblies /addmodule	Specify one or more modules to be part of this assembly.
/nostdlib	Do not import standard library (mscorlib.dll).
/reference	Import metadata from a file that contains an assembly.
Debugging/Error Checking	
/bugreport	Create a file that contains information that makes it easy to report a bug.
/checked	Specify whether integer arithmetic that overflows the bounds of the data type will cause an exception at run time.
/debug	Emit debugging information.
/fullpaths	Specify the absolute path to the file in compiler output.
/nowarn	Suppress the compiler's ability to generate specified warnings.
/warn	Set warning level.
/warnaserror	Promote warnings to errors.

Continued...

Option	Purpose
Preprocessor	
/define	Define preprocessor symbols.
Resources /linkresource	Link a .NET resource to an assembly.
/resource	Embed a .NET resource into the output file.
/win32icon	Insert a .ico file into the output file.
/win32res	Insert a Win32 resource into the output file.
Miscellaneous @	Specify a response file.
/?	List compiler options to stdout.
/baseaddress	Specify the preferred base address at which to load a DLL.
/codepage	Specify the code page to use for all source code files in the compilation.
/help	List compiler options to stdout.
/incremental	Enable incremental compilation of source code files.
/main	Specify the location of the Main method.
/nologo	Suppress compiler banner information.
/recurse	Search subdirectories for source files to compile.
/unsafe	Compile code that uses the unsafe keyword.

4.21 RuntimeTime Arguments

In this example, the program takes one argument at run time, converts the argument to a long number, and calculates the factorial of the number. If no arguments are supplied, the program issues a message that explains the correct usage of the program.

Listing 4.20 Factorial of a Number

```
// Command line arguments (cline.cs)
using System; public class Factorial {
   public static long Fac(long i)
   {
      return ((i <= 1) ? 1 : (i * Fac(i-1)));
   }
}
class MainClass {
   public static int Main(string[] args)
   {
      // Test if input arguments were supplied:
      if (args.Length == 0)
      {
         Console.WriteLine("Please enter a numeric argument.");
         Console.WriteLine("Usage: Factorial <num>");
         return 1;
```

```
        // Convert the input arguments to numbers:
        long num = long.Parse(args[0]);
            Console.WriteLine("The Factorial of {0} is {1}.", num,
Factorial.Fac(num));
        return 0;
    }
}
```

Sample Runs

The following are two sample runs of the program assuming that the program name is Factorial.exe.

Run #1:

```
C:\>cline 5
The Factorial of 5 is 120.
```

Run #2:

Factorial Please enter a numeric argument. Usage: Factorial <num>

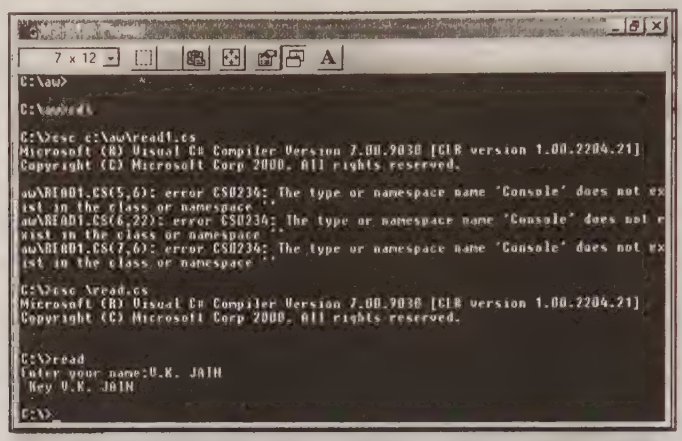
Reading

Listing 4.21

```
// Reading from matter entered at console command line (read.cs)
using System;
class ReadStr {
  public static void Main()
  {
     Console.Write("Enter your name:");
     string szName = Console.ReadLine();
     Console.WriteLine(" Hey "+ szName);
  }
}
```

Save this program as read.cs and compile it from commandline in DOS. This will create read.exe file when it compiles properly. Enter read to run the read.exe file at C: prompt and enter your name to see the output something similar to following:

Hey V.K. JAIN


C:\aw>
C:\aw\read1
C:\>csc c:\aw\read1.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
aw\READ1.CS(5,6): error CS0234: The type or namespace name 'Console' does not ex
ist in the class or namespace ''
aw\READ1.CS(6,22): error CS0234: The type or namespace name 'Console' does not e
xist in the class or namespace ''
aw\READ1.CS(7,6): error CS0234: The type or namespace name 'Console' does not ex
ist in the class or namespace ''
C:\>csc \read.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>read
Enter your name:V.K. JAIN
Hey V.K. JAIN
C:\>

4.22 Command-line arguments Samples

Sample Command Lines

```
1. csc File.cs
```

Compiles File.cs producing File.exe:

```
2. csc /target:library File.cs
```

Compiles File.cs producing File.dll:

```
3. csc /target:library File.cs
4. csc /nooutput
```

File.cs Compiles File.cs but no executable is created (useful for simply checking syntax):

```
5. csc /out:My.exe File.cs
```

Compiles File.cs and creates My.exe:

```
6. csc /define:DEBUG /optimize /out:File2.exe *.cs
```

Compiles all of the C# files in the current directory, with optimizations on and defines the DEBUG symbol. The output is File2.exe:

```
7. csc /target:library /out:File2.dll /warn:0 /nologo /debug *.cs
```

Compiles all of the C# files in the current directory producing a debug version of File2.dll. No logo and no warnings are displayed:

```
8. csc /target:library /out:Something.xyz *.cs
```

Compiles all of the C# files in the current directory to Something.xyz (a DLL):

In line 007, the Length property is used to obtain the length of the array. Notice that L

TYPES in C#

This lesson introduces C# expressions, types, and variables. It's objectives are:

1. To understand what a 'variable' is.
2. To learn the C# simple types.
3. To obtain a basic understanding of C# expressions.
4. To learn what the 'string type' is.

In simple words, *variables* are storage locations for data. You may place data into them and retrieve their contents as part of a C# expression. The interpretation of the data in a variable is controlled by *types*.

C# is a strongly 'Typed' language. Thus all operations on a variable are performed in consideration of what the variable's 'Type' is. There are rules that define what operations are legal in order to maintain the integrity of the data you put in a variable.

5.1 Types

The types of the C# language may be broadly divided into the following two categories:

1. value-type; and
2. reference-type.

A third category of types, *pointers*, is available but, in unsafe code.

The main difference between value types and reference types is that the variables of the value types directly contain their data, whereas variables of the reference types store references to their data — the latter being called objects. With reference types, it is possible for two variables to refer to the same object, and so the operations on one variable affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and hence, operations on one do not affect the other.

5.2 Value Types

A value type is either a struct type or an enumeration type. C# provides a set of predefined struct types called the simple types. The simple types are identified through reserved words, and are further subdivided into numeric types, integral types, and floating-point types.

value-type:
- struct-type
- enum-type

struct-type:
- type-name
- simple-type

simple-type:
- numeric-type
- bool

numeric-type:
- integral-type
- floating-point-type
- decimal

integral-type:
- sbyte
- byte
- short
- ushort
- int
- uint
- long
- ulong
- char

floating-point-type:
- float
- double

enum-type:
- type-name

All value types implicitly inherit from the class object. No type can derive from a value type, and thus the value types are implicitly sealed.

A variable of a value type always contains a value of that type. Unlike reference types, it is not possible for the value of a value type to be null or to reference an object of the further derived types. Assignment to a variable of a value type creates a copy of the value being assigned whereas assignment to a variable of a reference type, copies the reference, but not the object identified by the reference.

5.3 Default Constructors

All value types implicitly declare a public parameterless constructor called the default constructor. The default constructor returns a zero-initialized instance known as the default value for the value type:

For all simple-types, the default value is the value produced by a bit pattern of all zeros:

- For sbyte, byte, short, ushort, int, uint, long, and ulong, the default value is 0.
- For char, the default value is '\x0000'.
- For float, the default value is 0.0f.
- For double, the default value is 0.0d.
- For decimal, the default value is 0.0m.
- For bool, the default value is false.
- For an enum-type E, the default value is 0.
- For a struct-type, the default value is the value produced by setting all value type fields to their default value and all reference type fields to null.

5.4 Struct Types

A struct type is a value type that can declare constructors, constants, fields, methods, properties, indexers, operators, and nested types. Struct types are described in 11.

5.4.1 Simple Types

The C Sharp language provides a set of predefined struct types called the simple types. The simple types are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the System namespace, as described in the table 5.1.

Table 5.1 Predefined (struct)

Simple	Type	Reserved word	Aliased type
sbyte	System.SByte byte	System.Byte short	System.Int16
ushort	System.UInt16 int	System.Int32 uint	System.UInt32
long	System.Int64 ulong	System.UInt64 char	System.Char
float	System.Single double	System.Double bool	System.Boolean
decimal	System.Decimal		

Because a simple type aliases a struct type, every simple type has members. For example, int has the members declared in System.Int32 and the members inherited from System.Object, and the following statements are permitted:

```
int i = int.MaxValue;
// System.Int32.MaxValue constant string s = i.ToString();
// System.Int32.ToString() instance method string t = 123.ToString();
// System.Int32.ToString() instance method
```

The simple types permit some additional operations which other struct types do not. Following are the additional operations:

1. Most simple types permit values to be created by writing literals. For example, 123 is a literal of type int and 'a' is a literal of type char. C# makes no provision for literals of other struct types, and values of other struct types are ultimately always created through constructors of those struct types.
2. When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression at compile-time. Such an expression is known as a constant-expression. Expressions involving operators defined by other struct types always imply run time evaluation.
3. Through const declarations, it is possible to declare constants of the simple types. It is not possible to have constants for other struct types, only that a similar effect is provided by static readonly fields.
4. Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator.

5.4.2 Integral Types

C# supports nine integral types.They are sbyte, byte, short, ushort, int, uint, long, ulong, and char. The integral types have the following sizes and ranges of values:

- The sbyte type represents signed 8-bit integers with values between -128 and 127.
- The byte type represents unsigned 8-bit integers with values between 0 and 255.
- The short type represents signed 16-bit integers with values between -32768 and 32767.
- The ushort type represents unsigned 16-bit integers with values between 0 and 65535.
- The int type represents signed 32-bit integers with values between -2147483648 and 2147483647.
- The uint type represents unsigned 32-bit integers with values between 0 and 4294967295.
- The long type represents signed 64-bit integers with values between 9223372036854775808 and 9223372036854775807.
- The ulong type represents unsigned 64-bit integers with values between 0 and 18446744073709551615.
- The char type represents unsigned 16-bit integers with values between 0 to 65535. The set of possible values for the char type corresponds to the Unicode character set.

The integral-type unary and binary operators always operate with a signed 32-bit precision, an unsigned 32-bit precision, a signed 64-bit precision, or an unsigned 64-bit precision:

- For the unary + and ~ operators, the operand is converted to type T, where T is the first of int, uint, long, and ulong that can fully represent all possible values of the operand. The operation is then performed using the precision of type T, and the type of the result is T.
- For the unary-operator, the operand is converted to type T, where T is the first of int and long that can fully represent all possible values of the operand. The operation is then performed using the precision of type T, and the type of the result is T. The unary-operator cannot be applied to operands of type ulong.

- For the binary +, -, *, /, %, &, ^, |, ==, !=, >, <, >=, and <= operators, the operands are converted to type T, where T is the first of int, uint, long, and ulong that can fully represent all possible values of each operand. The operation is then performed using the precision of type T, and the type of the result is T (or bool for the relational operators). It is not possible for one operand involved with binary operators to be of the type long and the other to be of type ulong.
- For the binary << and >> operators, the left operand is converted to type T, where T is the first of int, uint, long, and ulong that can fully represent all possible values of the operand. The operation is then performed using the precision of type T, and the type of the result is T.

The char type is classified as an integral type, but it differs from the other integral types in the following two ways:

- There are no implicit conversions from other types to the char type. In particular, even though the sbyte, byte, and ushort types have ranges of values that are fully representable using the char type, implicit conversions from sbyte, byte, or ushort to char do not exist.
- Constants of the char type must be written as character-literals. Character constants can only be written as integer- literals in combination with a cast. For example, (char)10 is the same as '\x000A'.

The checked and unchecked operators and statements are used to control overflow check for integral-type arithmetic operations and conversions. In a checked context, an overflow produces a compile-time error or causes an OverflowException to be thrown. In an unchecked context, overflows are ignored and any high-order bits that do not fit in the destination type are simply discarded.

5.4.3 Floating Point Types

C# supports two floating point types—float and double. The float and double types are represented respectively using the 32-bit single- precision and the 64-bit double-precision IEEE 754 formats, which provide the following sets of values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two.
- Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero. For example 1.0 / 0.0 yields positive infinity, and -1.0 / 0.0 yields negative infinity.
- The Not-a-Number value, often abbreviated NaN. NaN's are produced by invalid floating-point operations, such as dividing zero by zero.
- The finite set of non-zero values of the form s x m x 2e, where s is 1 or -1, and m and e are determined by the particular floating-point type: For float, 0 < m < 224 and - 149 = e = 104, and for double, 0 < m < 253 and -1075 = e = 970.

The float type can represent values ranging from approximately 1.5 x 10-45 to 3.4 x 1038 precise upto 7 digits.

The double type can represent values ranging from approximately 5.0 x 10-324 to 1.7 x 10308 with a precision of 15-16 digits.

If one of the operands of a binary operator is of a floating- point type, then the other operand must be of an integral type or a floating-point type, and the operation is evaluated as follows:

- If one of the operands of is of an integral type, then that operand is converted to the floating-point type of the other operand.
- Then, if either of the operands is of type double, the other operand is converted to double, the operation is performed using at least double range and precision, and the type of the result is double (or bool for the relational operators).
- Otherwise, the operation is performed using at least float range and precision, and the type of the result is float (or bool for the relational operators).

The floating-point operators, including the assignment operators, never produce exceptions. Instead, in exceptional situations, floating-point operations produce zero, infinity, or NaN, which are as follows:

- If the result of a floating-point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a floating-point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a floating-point operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

Floating-point operations may be performed with higher precision than the result type of the operation. For example, some hardware architectures support an 'extended' or 'long double' floating- point type with greater range and precision than the double type, and uses them to implicitly perform all floating-point operations. Only at excessive cost in performance can such hardware architectures be made to perform floating-point operations with less precision, and rather than require an implementation to forfeit both performance and precision, C# allows a higher precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form x * y / z, where the multiplication produces a result that is outside the double range, but the subsequent division brings the temporary result back into the double range, the fact that the expression is evaluated in a higher range format may cause a finite result to be produced instead of an infinity.

5.4.4 The Decimal Type

The decimal type is a 128-bit data type suitable for financial and monetary calculations. The decimal type can represent values ranging from 1.0 x 10-28 to approximately 7.9 x 1028 with 28-29 significant digits.

The finite set of values of type decimal are of the form s x m x 10e, where s is 1 or -1, 0 = m < 296, and -28 = e = 0. The decimal type does not support signed zeros, infinities, or NaN's. A decimal is represented as a 96-bit integer scaled by a power of ten. For decimals with an absolute value less than 1.0m, the value is exact to the 28th decimal place, but no further. For decimals with an absolute value greater than or equal to 1.0m, the value is exact to 28 or 29

digits. Unlike the float and double data types, decimal fractional numbers such as 0.1 can be represented exactly in the decimal representation. In the float and double representations, such numbers are often infinite fractions, making those representations more prone to round-off errors.

If one of the operands of a binary operator is of type decimal, then the other operand must be of an integral type or of type decimal. An integral type operand if present, is converted to decimal before performing the operation.

Operations on values of type decimal are exact to 28 or 29 digits, but not beyond 28 decimal places. Results are rounded to the nearest representable value, and when a result is equally close to two representable values, to the value that has an even number in the least significant digit position. If a decimal arithmetic operation produces a value that is too small for the decimal format after rounding, the result of the operation becomes zero. If a decimal arithmetic operation produces a result that is too large for the decimal format, an OverflowException is thrown.

The decimal type has greater precision but smaller range than the floating-point types. Thus,conversions from the floating-point types to decimal might produce overflow exceptions, and conversions from decimal to the floating-point types might cause loss of precision. For these reasons, no implicit conversions exist between the floating-point types and decimal, and without explicit casts, it is not possible to mix floating-point and decimal operands in the same expression.

The Table 5.2 shows the integral types, their size, and range.

Table 5.2: Range and site of Integral Types

Type	Size (in bits)
sbyte	8 -128 to 127
byte	8 0 to 255
short	16 -32768 to 32767
ushort	16 0 to 65535
int	32 -2147483648 to 2147483647
uint	32 0 to 4294967295
long	64 -9223372036854775808 to 9223372036854775807
ulong	64 0 to 18446744073709551615
char	16 0 to 65535

Integral types are well-suited for those operations involving whole number calculations. The char type is the exception, representing a single Unicode character. As you can see from the aforementioned table, you have a wide range of options to choose from depending on your requirements.

The following table 5.3 shows the floating point and decimal types, their size, precision, and range.Table 5.3:

Type	Size (in bits)	Precision Range
float	32	7 digits 1.5 x 10-45 to 3.4 x 1038
double	64	15-16 digits 5.0 x 10-324 to 1.7 x 10308
decimal	128	28-29 decimal places 1.0 x 10-28 to 7.9 x 1028

Floating point types are used to perform operations requiring fractional representations. However, for financial calculations,.the decimal type is the better choice.
Results are computed by building expressions. These expressions are built by combining variables and operators together into statements. Table 5.2 describes the allowable operators, their precedence, and associativity.

Table 5.4: Operator's Precedence and Assosiativity

Category	Operator(s)	Associativity
Primary	(x) x.y f(x) a[x] x++ x—	left new typeof sizeof checked unchecked
Unary	+ - ! ~ ++x —x (T)x	left
Multiplicative	* / %	left
Additive	+ -	left
Shift	<< >>	left
Relational	< > <= >=	is left
Equality	== !=	right
Logical	AND &	left
Logical	XOR ^	left
Logical	OR |	left
Conditional	AND &&	left
Conditional	OR ||	left
Conditional	?:	right
Assignment	= *= /= %= += -= <<= >>= &= ^= |=	right

Left associativity means that operations are evaluated from left to right. Right associativity means that operations are evaluated from right to left. Such assignment operators evaluate everything to the right before the result is placed into the variable on the left.

5.4.5 The Bool Type

The bool type represents boolean logical quantities. The possible values of type bool are true and false.

Listing 5.1 using System;

```
\\bool1.cs
class Booleans {
    public static void Main() {
        bool content = true;
        bool noContent = false;
  Console.WriteLine("It is {0} that C# programming language is easier
than C++ and java.", content);
        Console.WriteLine("The statement above is not {0}.",
noContent);
    }
}
```

In Listing 5.1, the boolean values are written to the console as a part of a sentence. The 'bool' type is simply either a true or false. The output of the above program is shown as follows:

```
C:\>book1
Paperback Book Titles:
 Java Server Pages
 Programming WAP,Wap servlets
 C# Programming
Average Paperback Book Price: $49.67
```

No standard conversions exist between bool and other types. In particular, the bool type is distinct and separate from the integral types, and a bool value cannot be used in place of an integral value and vice versa. In the C and C++ languages, a zero integral value or a null pointer can be converted to the boolean value false, and a non-zero integral value or a non-null pointer can be converted to the boolean value true. In C#, such conversions are accomplished by explicitly comparing an integral value to zero or explicitly comparing an object reference to null.

5.5 Enumeration Types

An enumeration type is a distinct type which has named constants. Every enumeration type has an underlying type, which may be byte, sbyte, short, ushort, int, uint, long or ulong. Enumeration types are defined through enumeration declarations.

5.6 Reference Types

A reference type could be a class type, an interface type, an array type, or a delegate type.

reference-type:

```
class-type
interface-type
array-type
delegate-type
```

class-type:

```
type-name
object
string
```

interface-type:

```
type-name
```

array-type:

```
non-array-type
rank-specifiers
```

non-array-type:

```
type
rank-specifiers:
rank-specifier
rank-specifiers
rank-specifier
rank-specifier:
[   dim-separatorsopt
dim-separators:
,
dim-separators ,
delegate-type:
type-name
```

A reference type value is a reference to an instance of the type—the latter being called an object. The special value null is compatible with all reference types and indicates the absence of an instance.

5.7 Class Types

A class type defines a data structure that contains data members (constants, fields, and events), function members (methods, properties, indexers, operators, constructors, and destructors),and nested types. Class types support inheritance, a mechanism whereby the derived classes can possess attributes from a base class in addition to having attributes of their own. Instances of class types are created using object-creation- expressions.

5.8 The Object Type

Every type in C# directly or indirectly derives from the object class type.Thus, it is the general base class of all other types.

The object keyword is simply an alias for the predefined System.Object class. Writing the keyword object is exactly the same as writing System.Object, and vice versa.

5.9 The String Type

The string type is a sealed class type that inherits directly from object. Instances of the string class represent Unicode character strings.

Values of the string type can be written as string literals. The string keyword is simply an alias for the predefined System.String class. Writing the keyword string is exactly the same as writing System.String, and vice versa.

5.10 Interface Types

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

5.11 Array Types

An array is a data structure that contains a number of variables.These are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type—the element type of the array.

5.12 Delegate Types

A delegate is a data structure that refers to a static method or to an object instance and an instance method of that object. The nearest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only refer static functions, a delegate can refer both static and instance methods. In the latter case, the delegate stores a reference to the method's entry point, as also a reference to the object instance for which to invoke the method.

5.13 Predefined Types

The pre-defined types of C#are much like those in C and C++.

The predefined reference types are object and string. The type object is the ultimate base type of all other types. The type string is used to represent Unicode string values. Values of type string are immutable.

The predefined value types include signed and unsigned integral types, floating point types, and the types bool, char, and decimal. The signed integral types are sbyte, short, int, and long; the unsigned integral types are byte, ushort, uint, and ulong; and the floating point types are float and double.

The bool type is used to represent boolean values: values that are either true or false. The inclusion of bool makes it easier to write self-documenting code, and also helps to eliminate the common C++ coding error of inadvertantly using "=" when "==" should have been used. In C#, the example

int i = _; F(i); if (i = 0) // Bug: the test should be (i == 0) G(); is invalid because the expression i = 0 is of type int, and if statements require an expression of type bool.

The char type is used to represent Unicode characters. A variable of type char represents a single 16-bit Unicode character.

The decimal type is appropriate for calculations in which rounding errors caused by floating point representations are unacceptable. Common examples include financial calculations such as tax computations and currency conversions. The decimal type provides 28 significant digits.

The table 5.5 lists the predefined types, and shows how to write literal values for each of them.

Table 5.5 Predefined Types

Type	Description	Example
object	The ultimate base type of all other types	object o = null;
string	String type; a string is a sequence of Unicode characters	string s = "hello";
sbyte	8-bit signed integral type	sbyte val = 12;
short	16-bit signed integral type	short val = 12;
int	32-bit signed integral type	int val = 12;
long	64-bit signed integral type	long val1 = 12; long val2 = 34L;
byte	8-bit unsigned integral type	byte val1 = 12; byte val2 = 34U;
ushort	16-bit unsigned integral type	ushort val1 = 12; ushort val2 = 34U;
uint	32-bit unsigned integral type	uint val1 = 12; uint val2 = 34U;

ulong	64-bit unsigned integral type	ulong val1 = 12;
		ulong val2 = 34U;
		ulong val3 = 56L;
		ulong val4 = 78UL;

float	Single-precision floating point type	float val = 1.23F;
double	Double-precision floating point type	double val1 = 1.23; double val2 = 4.56D;
bool	Boolean type; a bool value is either true orfalse	bool val1 = true; bool val2 = false;
char character	Character type; a char value is a Unicode	char val = 'h';
decimal	Precise decimal type with 28 significant digits	decimal val = 1.23M;

The predefined types are simply shorthands for a system-provided type. For example, the keyword int refers to the struct System.Int32. As a matter of style, use of the keyword is favored over the use of the complete system type name.

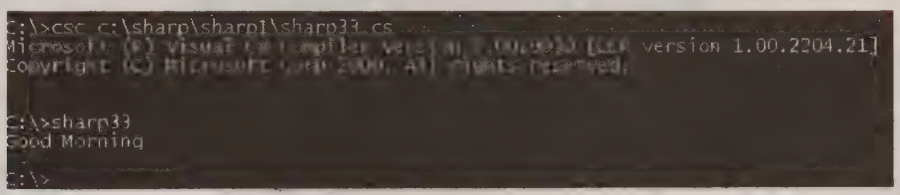
Predefined value types such as int are treated specially in a few ways but are for the most part treated exactly like other structs. Operator overloading enables developers to define new struct types that behave much like the predefined value types. For instance, a Digit struct can support the same mathematical operations as the predefined integral types, and can define conversions between Digit and predefined types.

Listing 5.2:Outputting Data Types

//sharp33.cs

```
class xyz33 {
   static void Main()
{
   string s;
   s="Good Morning";
   System.Console.WriteLine(s);
} }
```

The output of this program shall be displayed on console as:

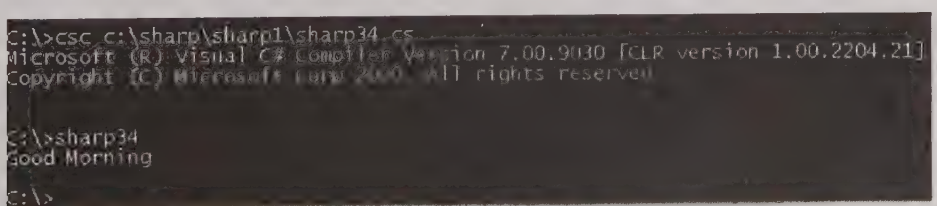


This program is an example of string type data represented by s. Take care to enclose the entire string within " "

Listing 5.3

```
//sharp34.cs class xyz34 {
   static void Main()
{
   string s;
   s="Good Morning";
   System.Console.WriteLine("{0}",s);
} }
```

The output of this program shall be displayed on console as:

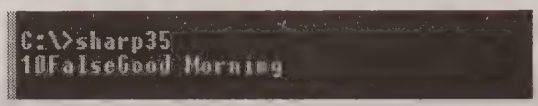

C:\>csc c:\sharp\sharp1\sharp34.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>sharp34
Good Morning
C:\>

Program sharp34.cs is a variation over program sharp33.cs .We have added {0} in WriteLine statement.This helps intake of different data types.

Listing 5.4

```
//sharp35.cs
class xyz35 {
   static void Main()
{
   string s;
   int integer1;
   bool boolean1;
   integer1=10;
   boolean1= false;
   s= "Good Morning";
   System.Console.WriteLine("{0}{1}{2}",integer1,boolean1,s);
} }
```

The output of this program shall be displayed on the console as:



Here, it should be noted that the predefined types employ operator overloading by themselves. For example, the comparison operators == and != have different semantics for different predefined types:

- Two expressions of type int are considered equal if they represent the same integer value.
- Two expressions of type object are considered equal if both refer to the same object, or if both are null.
- Two expressions of type string are considered equal if the string instances have identical lengths and identical characters in each character position, or if both are null.

Listing 5.5

```
class Test {
     static void Main() {
          string s = "Test";
          string t = string.Copy(s);
          Console.WriteLine(s == t);
          Console.WriteLine((object)s == (object)t);
     }
}
```

The above example produces the output:

because the first comparison is between two expressions of the type string, and the second is between the two expressions of the type object.

5.14 The concept of Boxing and Unboxing

Boxing and unboxing are central issues in C#'s type system. They provide a link between value-types and reference-types by permitting any value of a value-type to be converted to and from type object. This lends a unified view of the type system wherein a value of any type can ultimately be treated as an object. Example 5.5 clarifies the difference.

Listing 5.6

```
// (value1.cs)
using System;
class Class1 {
     public int Value = 0;
} class Test {
     static void Main() {
          int val1 = 0;
          int val2 = val1;
          val2 = 321;
          Class1 ref1 = new Class1();
          Class1 ref2 = ref1;
          ref2.Value = 321;
          Console.WriteLine("Values: {0}, {1}", val1, val2);
         Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
     }
}
```

The output of this program is

```
C:\>value1
Values: 0, 321
Refs: 321, 321
```

The assignment to the local variable val1 does not affect the local variable val2 because both local variables are of a value type (the type int) and each has its own storage. In contrast, note that the assignment ref2.Value = 321; affects the object that both ref1 and ref2 reference.

The lines

```
Console.WriteLine("Values: {0}, {1}", val1, val2);
Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
```

demonstrate some of the string formatting behavior of Console.WriteLine, which in fact takes a variable number of arguments. The first argument is a string, which may contain numbered placeholders like {0} and {1}. Each placeholder refers to a trailing argument with {0} referring to the second argument, {1} referring to the third argument, and so on. Before sending the output to the console,each placeholder is replaced with the formatted value of its corresponding argument. Developers can define new value types through enum and struct declarations, and new reference types via class, interface, and delegate declarations.

Listing 5.7

```
using System; public enum Color {
    Red, Blue, Green
} public struct Point {
    public int x, y;
} public interface IBase {
    void F();
} public interface IDerived: IBase {
    void G();
} public class A {
    protected virtual void H() {
        Console.WriteLine("A.H");
    }
} public class B: A, IDerived {
    public void F() {
        Console.WriteLine("B.F, implementation of IDerived.F");
    }
    public void G() {
        Console.WriteLine("B.G, implementation of IDerived.G");
    }
    override protected void H() {
        Console.WriteLine("B.H, override of A.H");
    }
}
public delegate void EmptyDelegate();
```

Listing 5.6 shows an example or two for each kind of type declaration. The later sections describe the type declarations in greater detail.

5.15 Conversions

C# supports two types of conversions—Implicit conversions and explicit conversions.

Implicit conversions are direct conversion. For example:

```
int iVal = 34; long lVal = intValue;
```

Explicit conversions includes type casting.conversion. For example:

```
long lVal = 123456; int iVal = (int) lVal;
```

C# allows you to define unary (+, -, !, ~, ++, —, true, or false), binary (+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, or <=) and conversion operators. There are differences to what you can do in C++ with these operators (for example, the unary dereference operator does not exist because there are no pointers), but the definition is similar.

```
class A
{
   public int val = 0;
   public static bool operator !(A a)
```

```
    {
        return a.val != 0;
    }
    public static bool operator !=(A a, A b)
    {
        return a.val != b.val;
    }
    public static bool operator ==(A a, A b)
    {
        return a.val == b.val;
    }
}
```

Here, we have defined an operator ! to test if an object has a zero value, and != for comparing two objects of the same type; the first being a unary operator, and the second a binary operator. Because we have defined the != operator, the compiler requires that we also define the == operator.

Unlike C++, C# cannot define an assignment operator; one has to use a conversion operator. C# offers two options for conversion—implicit and explicit. Implicit conversions occur in assignments whereas explicit conversions occur when you explicitly cast from one type to another:

```
A a = new A(10); int i = (int)a; // explicit a = 15;        // implicit
```

The code for these conversions is:

```
class A {
    public int val = 0;
    public A(int x)
    {
        val = x;
    }
    public static explicit operator int(A a)
    {
        return a.val;
    }
    public static implicit operator A(int a)
    {
        return new A(a);
    }
}
```

The predefined types also have predefined conversions. For instance, conversions exist between the predefined types int and long. Implicit conversions are supplied for conversions that can safely be performed without careful scrutiny. For instance, the conversion from int to long is an implicit conversion. This conversion always succeeds, and never results in a loss of information. Implicit conversions can be performed implicitly, as shown in the listing 5.8.

Listing 5.8

```
using System; class Test {
     static void Main() {
          int intValue = 123;
          long longValue = intValue;
          Console.WriteLine("{0}, {1}", intValue, longValue);
     }
}
```

This program implicitly converts an int to a long.
The output will be:

```
123, 123
```

In contrast, explicit conversions are performed with a cast expression.

Listing 5.9

```
using System; class Test {
     static void Main() {
          long longValue = Int64.MaxValue;
          int intValue = (int) longValue;
          Console.WriteLine("(int) {0} = {1}", longValue, intValue);
     }
}
```

The above code block explicitly converts a long to an int. The output is:

```
(int) 9223372036854775807 = -1
```

because an overflow occurs. Cast expressions permit the use of both implicit and explicit conversions.

5.16 Array Types

Please refer to unit 7 of this book for a detailed discussion on this topic.

5.17 Type System Unification

C# provides a 'unified type system'. All types-including value types-derive from the type object. It is possible to call object methods on any value, even values of 'primitive' types such as int.

The example

```
using System;
class Test {
     static void Main() {
          Console.WriteLine(3.ToString());
     }
}
calls the object-defined ToString method on an integer literal.'
```

Listing 5.10

```
class Test {
     static void Main() {
          int i = 123;
          object o = i;        // boxing
          int j = (int) o;     // unboxing
     }
}
```

is interesting. An int value can be converted to object and back again to int—by boxing and unboxing. When a variable of a value type needs to be converted to a reference type, an object box is allocated to hold the value, and the value is copied into the box. Unboxing is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location.

This type system unification provides value types with the benefits of object-ness without unnecessary overhead. For programs that don't need int values to act like objects, int values are simply 32-bit values. For programs that need int values to behave like objects,this feature may be availed. This ability to treat value types as objects bridges the gap between value types and reference types that exists in most languages. For example, a Stack class can provide Push and Pop methods that take and return object values.

```
public class Stack {
     public object Pop() {_}
     public void Push(object o) {_}
}
```

Because C# has a unified type system, the Stack class can be used with elements of any type, including value types like int.

5.18 Variables and Parameters

Variables represent storage locations where specific variables can be stored depending on the variable type. Local variables are variables that are declared in methods, properties, or indexers. They are usually defined by specifying a type name and a declarator that specifies the variable name and an optional initial value, as in:

```
int a; int b = 1;
```

Local variable declaration can also include multiple declarators. The declarations of a and b can be rewritten as:

int a, b = 1; A variable must be assigned before its value can be obtained.

Listing 5.11

```
class Test {
     static void Main() {
          int a;
          int b = 1;
```

```
            int c = a + b;
            _
        }
    }
```

This listing is invalid because it attempts to use the variable 'a' before assigning a value to it. A field is a variable that is associated with a class or struct, or an instance of a class or struct. A field declared with the static modifier defines a static variable, and a field declared without this modifier defines an instance variable. A static field is associated with a type, whereas an instance variable is associated with an instance.

```
Listing 5.12
using System.Data; class Employee {
     private static DataSet ds;
     public string Name;
     public decimal Salary;
     _
}
```

The above example shows an Employee class that has a private static variable and two public instance variables.

Formal parameter declarations also define variables. There are four kinds of parameters: value parameters, reference parameters, output parameters, and parameter arrays.

A value parameter is used for "in" parameter passing, in which the value of an argument is passed into a method, and modifications of the parameter do not affect the original argument. A value parameter refers to its own variable,which is distinct from the variable of the corresponding argument. This variable is initialized by copying the value of the corresponding argument.

Listing 5.13

```
//(pre1.cs)
using System;
class Test {
     static void F(int p) {
          Console.WriteLine("p = {0}", p);
          p++;
     }
     static void Main() {
          int a = 1;
          Console.WriteLine("pre: a = {0}", a);
          F(a);
          Console.WriteLine("post: a = {0}", a);
     }
```

Listing 5.13 shows a method F that has a value parameter named p. The above block produces the output:

```
C:\>pre1
pre: a = 420
p = 420
post: a = 420
```

even though the value parameter p is modified.

A reference parameter is used for "by reference" parameter passing, in which the parameter acts as an alias for a caller- provided argument. A reference parameter does not by itself define a variable, but rather refers to the variable of the corresponding argument. Modifications of a reference parameter directly and immediately affect the corresponding argument. A reference parameter is declared with a ref modifier.

Listing 5.14

```
using System; class Test {
     static void Swap(ref int a, ref int b) {
          int t = a;
          a = b;
          b = t;
     }
     static void Main() {
          int x = 1;
          int y = 2;
          Console.WriteLine("pre: x = {0}, y = {1}", x, y);
          Swap(ref x, ref y);
          Console.WriteLine("post: x = {0}, y = {1}", x, y);
     }
}
```

Listing 5.14 shows a Swap method that has two reference parameters. The output of the program is:

```
C:\>swap1
pre: x = 420, y = 100
post: x = 100, y = 420
```

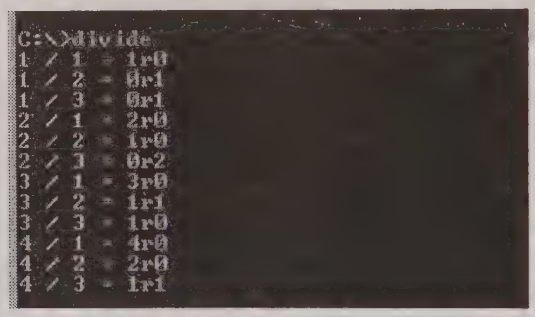
The ref keyword must be used for declaring the formal parameter as well as for using it. The use of ref at the call site calls special attention to the parameter so that a developer who reads the code will will be informed that the argument could change as a result of the call. An output parameter is similar to a reference parameter, except that the initial value of the caller-provided argument is unimportant. An output parameter is declared with an out modifier.

Listing 5.15

```
//(divide.cs)
using System;
class Test {
```

```
    static void Divide(int a, int b, out int result, out int remainder) {
            result = a / b;
            remainder = a % b;
        }
        static void Main() {
            for (int i = 1; i < 10; i++)
                for (int j = 1; j < 10; j++) {
                    int ans, r;
                    Divide(i, j, out ans, out r);
                      Console.WriteLine("{0} / {1} = {2}r{3}", i, j,
ans, r);
                }
        }
}
```

Listing 5.15 shows a Divide method that includes two output parameters-one for the result of the division and the other for the remainder.The result of this will be displayed as;



For value, reference, and output parameters, a one-to-one correspondence exists between caller-provided arguments and the parameters used to represent them. A parameter array enables a many-to-one relationship: many arguments can be represented by a single parameter array. In other words, parameter arrays enable variable length argument lists.

A parameter array is declared with a params modifier. There can be only one parameter array for a given method, and it is always the last parameter specified. The type of a parameter array is always a single dimensional array type. A caller can either pass a single argument of this array type, or any number of arguments of the element type of this array type. For instance, consider the listing 5.16

Listing 5.16

```
//(arg.cs)
using System;
class Test {
     static void F(params int[] args) {
          Console.WriteLine("# of arguments: {0}", args.Length);
          for (int i = 0; i < args.Length; i++)
          Console.WriteLine("\targs[{0}] = {1}", i, args[i]);
```

```
    static void Main() {
        F();
        F(1);
        F(1, 2);
        F(1, 2, 3);
        F(new int[] {1, 2, 3, 4});
    }
}
```

shows a method F that takes a variable number of int arguments, and several invocations of this method. The output is:

```
C:\>arg
# of arguments: 0
# of arguments: 1
        args[0] = 1
# of arguments: 2
        args[0] = 1
        args[1] = 2
# of arguments: 3
        args[0] = 1
        args[1] = 2
        args[2] = 3
# of arguments: 4
        args[0] = 1
        args[1] = 2
        args[2] = 3
        args[3] = 4
```

Most of the examples presented in this introduction use the WriteLine method of the Console class. The argument substitution behavior of this method, is illustrated in the ensuing example:

```
int a = 1, b = 2; Console.WriteLine("a = {0}, b = {1}", a, b);
```

The faculty is accomplished by using a parameter array. The WriteLine method provides several overloaded methods for the common cases in which a small number of arguments are passed, and one method that uses a parameter array.

```
namespace System {
    public class Console
    {
        public static void WriteLine(string s) {_}
        public static void WriteLine(string s, object a) {_}
        public static void WriteLine(string s, object a, object b)
{_}

        public static void WriteLine(string s, params object[] args)
{_}
    }
}
```

5.19 Automatic Memory Management

Manual memory management requires the allocation and de-allocation of memory blocks.This is a long-drawn and tedious affair. In C#, automatic memory management is provided so that developers are freed from this burdensome task. In the vast majority of cases, automatic memory management increases the code quality and enhances developer productivity without affecting either expressiveness or performance.

Listing 5.17

```
using System;
public class Stack {
     private Node first = null;
     public bool Empty {
          get {
               return (first == null);
          }
     }
     public object Pop() {
          if (first == null)
               throw new Exception("Can't Pop from an empty Stack.");
          else {
               object temp = first.Value;
               first = first.Next;
               return temp;
          }
     }
     public void Push(object o) {
          first = new Node(o, first);
     }
     class Node
     {
          public Node Next;
          public object Value;
          public Node(object value): this(value, null) {}
          public Node(object value, Node next) {
               Next = next;
               Value = value;
          }
     }
}
```

shows a Stack class implemented as a linked list of Node instances. Node instances are created in the Push method and are garbage collected when no longer needed. A Node instance becomes eligible for garbage collection when it is no longer possible for any code to access it. For instance, when an item is removed from the Stack, the associated Node instance becomes eligible for garbage collection.

Listing 5.18

```
class Test {
     static void Main() {
          Stack s = new Stack();
          for (int i = 0; i < 10; i++)
               s.Push(i);
          s = null;
     }
}
```

Listing 5.18 shows a test program that uses the Stack class. A Stack is created and initialized with 10 elements, and then assigned the value null. Once the variable s is assigned null, the Stack and the associated 10 Node instances become eligible for garbage collection. The garbage collector is permitted to clean up immediately, but is not required to do so.

The garbage collector underlying C# works by moving objects around in memory, but this motion is invisible to most C# developers. For developers who are generally content with automatic memory management but sometimes need fine-grained control or an extra edge in performance, C# provides the "unsafe" code. Such codes can deal directly with pointer types and object addresses. C# requires the programmer to fix objects to temporarily prevent the garbage collector from moving them.

This "unsafe" code feature is in fact a "safe" feature from the perspective of both developers and users. Unsafe code must be clearly marked in the code with the modifier unsafe, so developers can't possibly use unsafe language features accidentally, and the compiler and the execution engine work together to ensure that unsafe code cannot masquerade as safe code. These restrictions limit the use of unsafe code to situations in which the code is trusted.

Listing 5.19

```
//mem.cs
using System; class Test {
     unsafe static void WriteLocations(byte[] arr) {
          fixed (byte *p_arr = arr) {
               byte *p_elem = p_arr;
               for (int i = 0; i < arr.Length; i++) {
                    byte value = *p_elem;
                    string addr = int.Format((int) p_elem, "X");
       Console.WriteLine("arr[{0}] at 0x{1} is {2}", i, addr, value)
                    p_elem++;
               }
          }
     }
     static void Main() {
          byte[] arr = new byte[] {1, 2, 3, 4, 5};
          WriteLocations(arr);
     }
}
```

Listing 5.19 shows an unsafe method named WriteLocations that fixes an array instance and uses pointer manipulation to iterate over the elements. The index, value and location of each array element is written to the console. One possible output of the program is:

```
C:\>mem
arr[0] at 0xB4F670 is 1
arr[1] at 0xB4F671 is 2
arr[2] at 0xB4F672 is 3
arr[3] at 0xB4F673 is 4
arr[4] at 0xB4F674 is 5
```

but of course the exact memory locations may be different in different executions of the program.

5.20 Operands

C# includes unary operators, binary operators, and one ternary operator. Table 5.6 shows the operators listed in the descending order of precedence.

Table 5.6 : Operator Precedence

Category	Operators
Primary	(x) x.y f(x) a[x] x++ x—
new	typeof sizeof checked unchecked
Unary	+ - ! ~ ++x —x (T)x
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >=
is Equality	== !=
Logical	AND/ Logical XOR/Logical OR & ^ |
Conditional	AND/ Conditional OR/
Conditional	&& | | ?:
Assignment	=*= /= %= += -= <<=>>=&=^=|=

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression x + y * z is evaluated as x + (y * z) because the * operator has higher precedence than the + operator. When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

Except for the assignment operators, all binary operators are left-associative, meaning that operations are performed from left to right. For example, x + y + z is evaluated as (x + y) + z. The assignment operators and the conditional operator (?:) are right-associative, meaning that operations are performed from right to left. For example, x = y = z is evaluated as x = (y = z).

Precedence and associativity can be controlled using parentheses. For example, x + y * z first multiplies y by z and then adds the result to x, but (x + y) * z first adds x and y and then multiplies the result by z.

5.21 Statements

C# borrows most of its statements directly from C and C++, though there are some significant additions and modifications. The table below lists the kinds of statements that can be used, and provides an example for each.

Statement Example

5.21.1 Statement lists and block statements

```
    static void Main() {
    F();
    G();
    {
        H();
        I();
    }
}
```

5.21.2 Labeled statements and goto statements

```
    static void Main(string[] args) {
    if (args.Length == 0)
        goto done;
    Console.WriteLine(args.Length);
done:
    Console.WriteLine("Done");
}
```

5.21.3 Local constant declarations

```
    static void Main() {
    const float pi = 3.14;
    const int r = 123;
    Console.WriteLine(pi * r * r);
}
```

5.21.4 Local variable declarations

```
    static void Main() {
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

5.21.5 Expression statements

```
    static int F(int a, int b) {
    return a + b;
}
    static void Main() {
    F(1, 2);  // Expression statement
}
```

5.21.6 if statements

```
    static void Main(string[] args) {
    if (args.Length == 0)
        Console.WriteLine("No args");
    else
        Console.WriteLine("Args");
}
```

5.21.7 switch statements

```
    static void Main(string[] args) {
    switch (args.Length) {
        case 0:
            Console.WriteLine("No args");
            break;
        case 1:
            Console.WriteLine("One arg ");
            break;
        default:
            int n = args.Length;
            Console.WriteLine("{0} args", n);
            break;
    }
}
```

5.21.8 while statements

```
    static void Main(string[] args) {
    int i = 0;
    while (i < args.length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

5.21.9 do statements

```
    static void Main() {
    string s;
    do { s = Console.ReadLine(); }
    while (s != "Exit");
}
```

5.21.10 for statements

```
    static void Main(string[] args) {
    for (int i = 0; i < args.length; i++)
        Console.WriteLine(args[i]);
}
```

5.21.11 foreach statements

```
    static void Main(string[] args) {
    foreach (string s in args)
        Console.WriteLine(s);
}
```

5.21.12 break statements

```
    static void Main(string[] args) {
    int i = 0;
    while (true) {
        if (i > args.Length)
            break;
        Console.WriteLine(args[i++]);
    }
}
```

5.21.13 continue statements

```
    static void Main(string[] args) {
    int i = 0;
    while (true) {
       Console.WriteLine(args[i++]);
       if (i > args.Length)
            continue;
       break;
    }
}
```

5.21.14 return statements

```
    static int F(int a, int b) {
    return a + b;
}
    static void Main() {
    Console.WriteLine(F(1, 2));
    return;
}
```

5.21.15 throw statements and try statements

```
    static int F(int a, int b) {
    if (b == 0)
        throw new Exception("Divide by zero");
    return a / b;
}
    static void Main() {
    try {
        Console.WriteLine(F(5, 0));
    }
    catch(Exception e) {
        Console.WriteLine("Error");
    }
}
```

5.21.16 checked and unchecked statements

```
    static void Main() {
    int x = 100000, y = 100000;
    Console.WriteLine(unchecked(x * y));
    Console.WriteLine(checked(x * y));  // Error
    Console.WriteLine(x * y);           // Error
}
```

5.21.17 lock statements

```
    static void Main() {
    A a = _
    lock(a) {
        a.P = a.P + 1;
    }
}
```

5.21.18 using statements

```
    using System;
    class Resource: IDisposable
```

```
{
    public void F() {
        Console.WriteLine("Resource.F");
    }
    public void Dispose() {_}
    -
}
    static void Main() {
    using(Resource r = new Resource()) {
        r.F();
    }
}
```

5.22 Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in it. C# is a type-safe language, and the C# compiler guarantees that values stored in variables are always of the appropriate type. The value of a variable can be changed through assignment or through use of the ++ and — operators.

A variable must be definitely assigned before its value can be obtained.

As described in the following sections, variables are either *initially assigned* or *initially unassigned*. An *initially assigned* variable has a well-defined initial value and is always considered definitely assigned. An *initially unassigned* variable has no initial value. For an initially unassigned variable to be considered definitely assigned at a certain location, an assignment to the variable must occur in every possible execution path leading to that location.

5.22.1 Variable Categories

C# defines seven categories of variables:

- static variables,
- instance variables,
- array elements,
- value parameters,
- reference parameters,
- output parameters, and
- local variables.

Each of these categories is described in the ensuing topic.

Listing 5.20

```
class A {
     public static int x;
     int y;
     void F(int[] v, int a, ref int b, out int c) {
          int i = 1;
          c = a + b++;
     }
}
```

In example 5.18, x is a static variable, y is an instance variable, v[0] is an array element, a is a value parameter, b is a reference parameter, c is an output parameter, and i is a local variable.

5.22.2 Static Variables

A field declared with the static modifier is called a static variable. A static variable comes into existence when the type in which it is declared is loaded (10.11) and ceases to exist when the program terminates. The initial value of a static variable is the default value (5.2) of the variable's type.

For purposes of definite assignment checking, a static variable is considered initially assigned.

5.22.3 Instance Variables

A field declared without the static modifier is called an *instance* variable. The following points highlights the use of instance variables in Classes and Structs:

1. **Instance Variables in Classes**: An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's destructor (if any) has executed.

   The initial value of an instance variable of a class is the default value of the variable's type. For purposes of definite assignment checking, an instance variable of a class is considered initially assigned.

2. **Instance Variables in Structs**: An instance variable of a struct has exactly the same lifetime as the struct variable to which it belongs. In other words, when a variable of a struct type comes into existence or ceases to exist, so too do the instance variables of the struct.

The initial assignment state of an instance variable of a struct in the same as that of the containing struct variable. In other words, when a struct variable is considered initially assigned, so too are its instance variables, and when a struct variable is considered initially unassigned, its instance variables too are likewise unassigned.

5.22.4 Array Elements

The elements of an array come into existence when an array instance is created, and cease to exist when there are no references to that array instance.

The initial value of each of the elements of an array is the default value (5.2) of the type of the array elements. For purposes of definite assignment checking, an array element is considered initially assigned.

5.22.5 Value Parameters

A parameter declared without a ref or an out modifier is a value parameter.

A value parameter comes into existence upon invocation of the function member (method, constructor, accessor, or operator) to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter ceases to exist upon return of the function member. For purposes of definite assignment checking, a value parameter is considered initially assigned.

5.22.6 Reference Parameters

A parameter declared with a ref modifier is a reference parameter.

A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of a reference parameter is always the same as the underlying variable.

The following definite assignment rules apply to reference parameters.

- A variable must be definitely assigned before it can be passed as a reference parameter in a function member invocation.
- Within a function member, a reference parameter is considered initially assigned.

Within an instance method or instance accessor of a struct type, the keyword behaves exactly as a reference parameter of the struct type.

5.22.7 Output Parameters

A parameter declared with an out modifier is an output parameter. An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of an output parameter is always the same as the underlying variable.

The following definite assignment rules apply to output parameters. Please observe the different rules for reference parameters described in.

- A variable need not be definitely assigned before it can be passed as an output parameter in a function member invocation.
- Following a function member invocation, each variable that was passed as an output parameter is considered assigned in that execution path.
- Within a function member, an output parameter is considered initially unassigned.
- Every output parameter of a function member must be definitely assigned before the function member returns.

Within a constructor of a struct type, the keyword behaves exactly as an output parameter of the struct type.

5.22.8 Local Variables

A local variable is declared by a local-variable-declaration, which may occur in a block, a for-statement, a switch-statement, or a using-statement. A local variable comes into existence when the control enters the block, for-statement, switch-statement, or the using-statement that immediately contains the local variable declaration. A local variable ceases to exist when control leaves its immediately containing block, for-statement, switch- statement, or using-statement. A local variable is not automatically initialized and thus has no default value. For purposes of definite assignment checking, a local variable is considered initially unassigned. A local-variable-declaration may include a variable-initializer, in which case the variable is considered definitely assigned in its entire scope, except within the expression provided in the variable-initializer. Within the scope of a local variable, it is an error to refer to the local variable in a textual position that precedes its variable-declarator.

5.23 Expressions

An *expression* is a sequence of operators and operands that specifies a computation.

5.23.1 Expression Classifications

An expression is classified as one of the following:

- A value.
- A variable.
- A namespace.
- A type.
- A method group
- A property access.
- An event access.
- An indexer access.

1. The value of a variable is simply the value currently stored in the storage location identified by the variable. A variable must be considered definitely assigned before its value can be obtained, or otherwise a compile-time error occurs.
2. The value of a property access expression is obtained by invoking the get-accessor of the property. If the property has no get-accessor, an error occurs. Otherwise, a function member invocation is performed, and the result of the invocation becomes the value of the property access expression.
3. The value of an indexer access expression is obtained by invoking the get-accessor of the indexer. If the indexer has no get-accessor, an error occurs. Otherwise, a function member invocation is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

5.24 Operators

Expressions are constructed with operands and operators. The operators of an expression indicate which operations to apply to the operands. Examples of operators include +, -, *, /, and new. Examples of operands include literals, fields, local variables and expressions.There are three types of operators:

1. **Unary operators**: The unary operators take one operand and use either prefix notation (such as -x) or postfix notation (such as x++).
2. **Binary operators:** The binary operators take two operands and all use infix notation (such as x + y).
3. **Ternary operator**: Only one ternary operator, ?:, exists. The ternary operator takes three operands and uses infix notation (c? x: y).

The order of evaluation of operators in an expression is determined by the precedence and associativity of the operators. Certain operators can be overloaded. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user- defined class or struct type.Thus over-loading permits customization of operators.

5.24.1 Unary Operators

Listing 5.21: Unary Operators

```
Unary1.cs
using System;
class Unary {
    public static void Main() {
        int unary = 0;
        int preIncrement;
        int preDecrement;
        int postIncrement;
        int postDecrement;
        int positive;
        int negative;
        sbyte bitNot;
        bool logNot;
        preIncrement = ++unary;
        Console.WriteLine("Pre-Increment: {0}", preIncrement);
        preDecrement = --unary;
        Console.WriteLine("Pre-Decrement: {0}", preDecrement);
        postDecrement = unary--;
        Console.WriteLine("Post-Decrement: {0}", postDecrement);
        postIncrement = unary++;
        Console.WriteLine("Post-Increment: {0}", postIncrement);
        Console.WriteLine("Final Value of Unary: {0}", unary);
        positive = -postIncrement;
        Console.WriteLine("Positive: {0}", positive);
        negative = +postIncrement;
        Console.WriteLine("Negative: {0}", negative);
        bitNot = 0;
        bitNot = (sbyte)(~bitNot);
        Console.WriteLine("Bitwise Not: {0}", bitNot);
        logNot = false;
        logNot = !logNot;
        Console.WriteLine("Logical Not: {0}", logNot);
    }
}
```

When evaluating expressions, post-increment and post-decrement operators return their current values before applying the operators. However, when using pre-increment and pre-decrement operators, the operator is applied to the variable prior to returning the final value.
In Listing 5.21, the "unary" variable is initialized to zero. When the pre-increment (++x) operator is used, "unary" is incremented to 1 and the value 1 is assigned to the "preIncrement" variable. The pre-decrement (—x) operator turns "unary" back to a 0 and then assigns the value to the "preDecrement" variable.

When the post-decrement (x—) operator is used, the value of "unary", 0, is placed into the "postDecrement" variable and then the "unary" is decremented to -1. Next the post increment (x++) operator moves the current value of "unary", -1, to the "postIncrement" variable and then increments "unary" to 0.

The variable "bitNot" is initialized to zero and the bitwise not operator is applied. The bitwise not (~) operator flips the bits in the variable. In this case, the binary representation of 0, "00000000", was transformed into -1, "11111111".

Oberve the expression "(sbyte)(~bitNot)". Any operation performed on types sbyte, byte, short, or ushort return integer values. To assign the result into the bitNot variable, we have to use a cast (Type) operator, where Type is the type you wish to convert to (in this case - sbyte). Cast operators must be performed explicity when you go from a larger type to a smaller type to guard against data loss. Generally speaking, assigning a smaller type to a larger type is risk-free since the larger type has room to hold the entire value. Take care against casting between signed and unsigned types. You mustalso ensure that the integrity of your data is preserved. Many basic programming texts contain good descriptions of bit representations of variables and the dangers of explicit casting.

The logical not (!) operator allows you to toggle the value of a boolean variable. In the example, the "logNot" variable is changed from false to true. You can expect the following output from the above program.

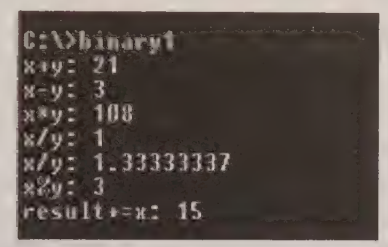
```
C:\>unary1
Pre-Increment: 1
Pre-Decrement: 0
Post-Decrement: 0
Post-Increment: -1
Final Value of Unary: 0
Positive: 1
Negative: -1
Bitwise Not: -1
Logical Not: True
```

Listing 5.22: Binary Operations

```
Binary1.cs
using System;
class Binary {
    public static void Main() {
        int x, y, result;
        float floatResult;
        x = 12;
        y = 9;
        result = x+y;
        Console.WriteLine("x+y: {0}", result);
        result = x-y;
        Console.WriteLine("x-y: {0}", result);
        result = x*y;
        Console.WriteLine("x*y: {0}", result);
```

```
            result = x/y;
            Console.WriteLine("x/y: {0}", result);
            floatResult = (float)x/(float)y;
            Console.WriteLine("x/y: {0}", floatResult);
            result = x%y;
            Console.WriteLine("x%y: {0}", result);
            result += x;
            Console.WriteLine("result+=x: {0}", result);
        }
    }
```

The output of this program shall be displayed on console as:



Listing 5.22 shows several examples of binary operators. As you might expect, the results of addition (+), subtraction (-), multiplication (*), and division (/) produce the expected mathematical results.

The "floatResult" variable is a floating point type. We explicitly cast the integer variables "x" and "y" to calculate a floating point value.

There is also an example of the remainder(%) operator. It performs a division operation on two values and returns the remainder.

The last statement shows another form of the assignment with operation (+=) operator. Any time you use the assignment with the operation operator, it is the same as applying the binary operator to both the left hand and right hand sides of the operator and putting the results into the left hand side. The example could have been written as "result = result + x" and in that case too,it would have returned the same value.

One type you have come seen very often in the last two lessons is the "string" type. The "string" type is represented by a list of Unicode characters within single quotes. i.e. "This is a string."

Another data type is the Array. Arrays can be thought of as containers that have a list of storage locations for a specified type. When declaring an Array, you specify the type, Array name, dimensions, and size.

5.24.2 Operator Precedence and Associativity

When an expression contains multiple operators, the precedence of the operators control the order in which the individual operators are evaluated. For example, the expression x + y * z is evaluated as x + (y * z) because the * operator has higher precedence than the + operator. The precedence of an operator is established by the definition of its associated grammar production.

For example, an additive-expression consists of a sequence of multiplicative- expressions separated by + or - operators, thus giving the + and - operators lower precedence than the *, /, and % operators.

5.24.3 Operator Overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including operator declarations in classes and structs. User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered.

The overloadable unary operators are:

```
+   -   !   ~   ++   --   true   false
```

The overloadable binary operators are:

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=
<=
```

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the =, &&, ||, ?:, new, typeof, sizeof, and is operators. When a binary operator is overloaded, the corresponding assignment operator is also implicitly overloaded. For example, an overload of operator * is also an overload of operator *=. This will be described further . Note that the assignment operator itself (=) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable. Cast operations, such as (T)x, are overloaded by providing user- defined conversions.

Listing 5.23: Operator Overloading

Following program shows some of the overloadable operator.

```
//over11.cs
using System;
struct RomanNumeral
{
   public RomanNumeral(int value)
{
   this.value = value;
}
   public override string ToString()
{
   return(value.ToString());
}
   public static RomanNumeral operator -(RomanNumeral roman)
{
   return(new RomanNumeral(-roman.value));
}
   public static RomanNumeral operator +(
   RomanNumeral    roman1,
```

```
    RomanNumeral    roman2)
{
    return(new RomanNumeral(
                    roman1.value + roman2.value));
}
    public static RomanNumeral operator ++(
    RomanNumeral    roman)
{
    return(new RomanNumeral(roman.value + 1));
}
    int value;
}
    class Test
{
    public static void Main()
{
    RomanNumeral    roman1 = new RomanNumeral(12);
    RomanNumeral    roman2 = new RomanNumeral(125);

    Console.WriteLine("Increment: {0}", roman1++);
    Console.WriteLine("Addition: {0}", roman1 + roman2);
} }
```

The output of this program shall be displayed on console as:

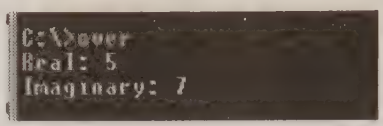
```
C:\>over11
Increment: 12
Addition: 138
```

Listing 5.24 Another Example of Operator Overloading

```
//over.cs
using System; public class Complex {
  public int real = 0;
  public int imaginary = 0;
public Complex(int real, int imaginary)
 {
   this.real = real;
   this.imaginary = imaginary;
 }
public static Complex operator +(Complex c1, Complex c2) {
   return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
}
public static void Main() {
 Complex num1 = new Complex(2,3);
 Complex num2 = new Complex(3,4);
 Complex sum = num1 + num2;
```

```
Console.WriteLine("Real: {0}", sum.real); Console.WriteLine("Imaginary:
{0}", sum.imaginary); } }
```

The following is the output;



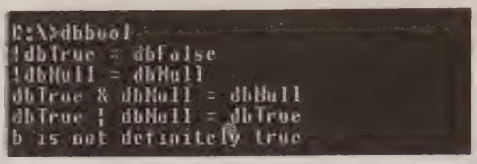
Listing 5.25

```
//dbbool.cs
using System;
public struct DBBool {
// The three possible DBBool values.
public static readonly DBBool dbNull = new DBBool(0);
public static readonly DBBool dbFalse = new DBBool(-1);
public static readonly DBBool dbTrue = new DBBool(1);
// Private field that stores -1, 0, 1 for dbFalse, dbNull, dbTrue.
int value;
// Private constructor. The value parameter must be -1, 0, or 1.
DBBool(int value) {
   this.value = value;
}
// Implicit conversion from bool to DBBool. Maps true to
// DBBool.dbTrue and false to DBBool.dbFalse. public static implicit
operator DBBool(bool x) {
   return x? dbTrue: dbFalse;
}
// Explicit conversion from DBBool to bool. Throws an
// exception if the given DBBool is dbNull, otherwise returns
// true or false.
public static explicit operator bool(DBBool x) {
  if (x.value == 0) throw new InvalidOperationException();
  return x.value > 0;
}
// Equality operator. Returns dbNull if either operand is dbNull,
// otherwise returns dbTrue or dbFalse.
public static DBBool operator ==(DBBool x, DBBool y) {
if (x.value == 0 || y.value == 0)
return dbNull; return x.value == y.value? dbTrue: dbFalse;
}
// Inequality operator. Returns dbNull if either operand is dbNull,
// otherwise returns dbTrue or dbFalse.
public static DBBool operator !=(DBBool x, DBBool y) { if (x.value == 0
|| y.value == 0) return dbNull; return x.value != y.value? dbTrue:
dbFalse; }
// Logical negation operator. Returns dbTrue if the operand is
```

```
// dbFalse, dbNull if the operand is dbNull, or dbFalse if the operand
// is dbTrue.
public static DBBool operator !(DBBool x) { return new DBBool(-x.value);
}
// Logical AND operator. Returns dbFalse if either operand is
// dbFalse, dbNull if either operand is dbNull, otherwise dbTrue.
public static DBBool operator &(DBBool x, DBBool y) { return new
DBBool(x.value < y.value? x.value: y.value); }
// Logical OR operator. Returns dbTrue if either operand is dbTrue,
// dbNull if either operand is dbNull, otherwise dbFalse. public static
DBBool operator |(DBBool x, DBBool y) {
  return new DBBool(x.value > y.value? x.value: y.value);
}
// Definitely true operator. Returns true if the operand is
// dbTrue, false otherwise.
public static bool operator true(DBBool x) {
   return x.value > 0;
}
// Definitely false operator. Returns true if the operand is
// dbFalse, false otherwise.
public static bool operator false(DBBool x) {
   return x.value < 0;
}
// Overloading the conversion from DBBool to string.
public static implicit operator string(DBBool x) { return x.value > 0 ?
"dbTrue" : x.value < 0 ? "dbFalse" : "dbNull"; }
// Override the ToString method to convert DBBool to a string.
public override string ToString() {
  return (string)this;
}
 }
class Test
{
   static void Main()
   {
         DBBool a, b;
         a = DBBool.dbTrue;
         b = DBBool.dbNull;
         Console.WriteLine( "!{0} = {1}", a, !a);
               Console.WriteLine( "!{0} = {1}", b, !b);
               Console.WriteLine( "{0} & {1} = {2}", a, b, a & b);
               Console.WriteLine( "{0} | {1} = {2}", a, b, a | b);
               if (b)
                     Console.WriteLine("b is definitely true");
               else
```

```
                Console.WriteLine("b is not definitely true");
        }
    }
```

The following is the output:



5.25 Boxing And Unboxing

Value types can be converted to objects which may then be used to perform type operations. This process is called boxing and the reverse- converting an object to a value type - is called unboxing. For example:

```
int a = 42;
object o = a;    // boxing
Console.WriteLine(o.GetType().ToString()); int b = (int)o; // unboxing
```

The process of boxing causes a copy to be made, so

```
b = 43; Console.WriteLine("{0} {1}", a, b);
```

results in 42 43 printed on the console.

When a value type is boxed, án object instance is allocated and the value of the value type is copied into the new object.

Look at the following example:

```
int nFunny = 2000; object oFunny = nFunny;
```

The assignment in the second line implicitly invokes a boxing operation.The value of the nFunny integer variable is copied to the object oFunny. Now both the integer variable and the object variable exist on the stack, but the value of the object resides on the heap.

This implies that the values are independent of each other-there is no link between them.(oFunny does not refrence the value of nFunny.) The following code illustrates the consequences:

```
int nFunny = 2000;
object oFunny = nFunny;
oFunny = 2001;
Console.WriteLine("{0} {1}", nFunny, oFunny);
```

When the code changes the value of oFunny, the value of nFunny is not changed. This functionality of value types can be exploited fully by a clear understanding of the copy behaviour.

5.25.1 Unboxing Conversions

Whereas boxing is an implicit operation-you have to tell the compiler which value type you want to extract from the object-unboxing is an implicit one. When performing the unboxing operation,

C# checks that the value type you request is actually stored in the object instance. Only then is the value unboxed.

This is how unboxing is performed:

```
int nFunny = 2000;
object oFunny = nFunny;
int nNotSofunny = (int)oFunny;
```

If you mistakenly requested a double value

```
double nNotSoFunny = (double)oFunny;
```

the NGWS runtime would raise an Invalid Exception exception.You can learn more about exception handling in Chapter ..,"Exception Handling."
Unboxing is thus, an explicit operation.You have to tell which value type you want to extract from the object.Consider the following code:

```
Int32 vt= 10; //value type variable
```

object rt= vt; /*memory is allocated on the heap of size equal to size of vt,the value type bits are copied to the newly allocated memory and the address of the object is returned and stored in rt.This is typically Boxing.*/ Int32 vt2=(Int32)rt;//Unboxing

What is needed to compile?

.NET SDK

How to Compile?

```
csc /r : System.dll /r:System.winforms.dll /r:Microsoft.win32.interop.dll
/r:System.Drawing.dll responsiveui.cs
```

Code:

```
using System;
= class BoxAndUnBox{
    public static void Main() {
    Int32 vt= 10;
 //value type variable
   object rt= vt; /*memory is allocated on the heap of size equal to
size of
vt,the value type bits are copied to the newly allocated memory and the
address of the object is returned and stored in rt.*/
```

This process is called Boxing. Now consider this example:

```
int a = 42;
 object o = a;
   // boxing Console.WriteLine(o.GetType().ToString());
 int b = (int)o;
 // unboxing
```

The process of boxing causes a copy to be made, so

```
b = 43;
Console.WriteLine("{0} {1}", a, b);
```

results is 42 43 printed on the console.
When a value type is boxed, an object instance is allocated and the value of the value type is copied into the new object.

Look at the following example:

```
int nFunny = 2000; object oFunny = nFunny;
```

The assignment in the second line implicitly invokes a boxing operation.The value of the nFunny integer variable is copied to the object oFunny. Now both the integer variable and the object variable exist on the stack, whereas, the value of the object resides on the heap.

5.26 Type Declarations

A type-declaration is either a/an

- class-declaration
- struct-declaration
- interface-declaration
- enum-declaration
- delegate-declaration

A type-declaration can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct. When a type declaration for a type T occurs as a top-level declaration in a compilation unit, the fully qualified name of the newly declared type is simply T. When a type declaration for a type T occurs within a namespace, class, or struct, the fully qualified name of the newly declared type is N.T, where N is the fully qualified name of the containing namespace, class, or struct. A type declared within a class or struct is called a nested type.

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place as explained :

1. Types declared in compilation units or namespaces can have public or internal access,the default being internal access.
2. Types declared in classes can have public, protected internal, protected, internal, or private access. The default is private access.
3. Types declared in structs can have public, internal, or private access,the default being private access.

5.26.1 Types Description

Value Types Include simple data types such as int, char, bool, enums etc.

Reference Types Includes object, class, interface, delegate, and array types.

Value Types- Value type objects direct contain the actual data in a variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

```
int i = 10;
```

Reference Types- Reference type variables stores the reference of the actual data. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable.

```
MyClass cls1 = new MyClass();
```

5.27 Value Data Types

The data types of C# are similar to those of C++, but have some marked differences too.For example, all variables are affected by the C# unified type system, even seemingly non-object types.In.NET every type is actually an object and is an instance of a class or struct. Every .NET class has System.Object is the ultimate base class in the class hierarchy which means that you can call the methods of the Object class through basic types like int.

C# makes the difference between value types and reference types. When you pass a variable as a function parameter, the actual value is passed to the function. When a reference type is passed it is not the actual value but a reference — an identifier that C# used to identify the actual variable — that is passed. When a function alters a parameter passed by value, it is the local copy that is altered:

```
void AddOne(int i) {
   i = i + 1;
   Console.WriteLine("AddOne: {0}", i);
}
void UseIt()
{
   int j = 10;
   Console.WriteLine("j = {0}", j);
   AddOne(j);
   Console.WriteLine("j = {0}", j);
}
```

The results from this code will be:

```
j = 10 AddOne: 11 j = 10
```

In other words, the value of j in UseIt() is unchanged by calling AddOne() because a copy of the value if j is passed to the function. However, the i parameter of AddOne() is changed, but this only has an effect in the function. To make a value type passed by reference you have to use the ref modifier in both the function declaration and the function call (I'll explain why later). For example,

```
void AddOne(ref int i)
{
   i = i + 1;
   Console.WriteLine("AddOne: {0}", i);
} void UseIt() {
   int j = 10;
   Console.WriteLine("j = {0}", j);
   AddOne(ref j);
   Console.WriteLine("j = {0}", j);
}
```

These will give the following results:
j = 10
AddOne: 11
j = 11

That is, the value that AddOne() acts on is j declared in UseIt() because j was passed by reference.
One reason for ref is to allow a method to return values through parameters when the function return value is used to return a status code.But this approach is not suitable when the method being called is on an object on another machine. In this case whenever i is accessed (3 times in AddOne() in the example above) a call is made across the network. C# offers the out specifier for situations like this. This indicates that the parameter is write-only:

```
void now(out String str) {
   // str does not exist at this point
   DateTime dt = DateTime.Now;
   str = dt.ToString();
}
```

The DateTime.Now property creates a new instance of the DateTime class initialized to the current date and time. The ToString() is a method inherited from System.Object that returns a string with the stringized form of the date. (Note that it would have been better for now() to return a String, but then I would not be able to make this point about out.) The use of the out modifier says that the parameter is created in the method and a reference is returned to the caller. now() is called like this:
String current; now(out current); Console.WriteLine(current);

Again, the calling code has to use the out modifier too. The reason is that functions can be overloaded with out and ref:

```
void f(A a) {/* a is passed in, by reference or by value
   depending on the type of A */}
void f(ref A a) {/* a is passed by reference and so is in/out */} void
f(out A a) {/* a is passed out by reference */}
```

The only way that the C# compiler can tell which overload of f() you want is if you specify it explicitly in the calling code. As mentioned earlier, when you create an object the variable you assign is a reference. Thus, when you pass an object via a method call and the parameter is changed, it means that so is the original object, for example:

```
class Q {
   public int i;
}
void AddOne(Q qq)
{
   qq.i = qq.i + 1;
   Console.WriteLine("AddOne: {0}", qq.i);
}
void UseIt()
```

```
{
    Q q = new Q();
    q.i = 10;
    Console.WriteLine("q.i = {0}", q.i);
    AddOne(q);
    Console.WriteLine("q.i = {0}", q.i);
}
```

In this case q is passed by reference (because it is an object) so AddOne() increments q rather than the local qq. The output of this code would be:

```
q.i = 10
```

AddOne: 11

```
q.i = 11
```

CLASSES

This lesson introduces you to C# Classes. Our objectives here are as follows:

- Implement Constructors.
- Know the difference between instance and static members.
- Using Destructors.
- Familiarization with Class Members.

6.1 C# Classes

Classes and derived classes are the most important and latest language concepts C# has introduced. Let us slowly unravel these new concepts. When we write a program we have to reduce the "real- life" problem to a model. That is easy for physicists. The process of modeling is called abstraction. The model defines an abstract representation of the problem, we are dealing with. Usually the model can be splitted into different parts of a set of data, which affects a set of operations and connects different pieces of data. In most cases, the model is also complicated and you can group your model into a hierarchy and define a subset of the data and a set of operations which only affects the subset of data as a new abstract data type (ADT). These operations often are called messages since they change the status of the data contained by the ADT. The operations and relations between these ADTs represent the model itself. Many of the design elements of C#—and object oriented languages in general—try to emulate the way we interact with the real world. For example, whenever you look around yourself you see a large number of objects. We organize all of the objects around us in our minds by arranging them in hierarchical categories, or "classes". For example, you have in your hands a book. A book is a general class of object. You might say, "This object I am holding is classified as a book."

A hierarchy of object classes surrounds the class "book", and it extends in two directions. Books are a member of a general class "publications". Specific types of books also exists: computer books, fiction books, biographical books, and so on. The hierarchy extends both towards the general and the specific. At this point you are holding a single, particular book. In OOPs lingo, you are holding an "instance" of the class "book".

Books have certain attributes that are shared by all books: They have a cover, several chapters, no advertising, and so on. They also have attributes shared by publications in general: a title, a date of publication, a publisher, etc. They have attributes that are shared by all physical objects: a location, size, shape, and weight. This idea of shared attributes is very important in C#. C# models the concept of shared attributes using inheritance.

There are certain things you do with and to different objects, and those actions change from object to object. For example, you can read a book, and you can flip its pages. You can look at the title, find a specific chapter, look something up in the index, count the number of pages, and so on. These actions are largely unique to publications: you never find yourself flipping the pages of a hammer, for example. However, there are actions that are generic to all physical objects, such as picking them up. C# takes this fact about the world into account as well, again using inheritance.

The hierarchical nature of object categories, as well as our hierarchical organization of object attributes and actions, are all embedded into the syntax and vocabulary of C#. For example, when designing a program you will break it down into objects, each of which has a "class". You will "inherit" features of a "base class" when you create a "derived class". That is, you will create general object classes and then make more specific classes from them, deriving the particulars from the general classes. You will "encapsulate" the data found in an object with "member functions", and as you extend a class, you will "overload" and "override" the functions of the base class. It may not be very clear to you. Let us look at a quick example to see what all of these words actually mean.

The classic example of object oriented programming is a graphics program that allows you to draw objects—lines, rectangles, circles and such—on the screen. What do all of these objects share in common? All objects have a location on the screen. They might also have a color. These attributes are possessed by every shape shown on the screen. Therefore, as a program designer you would create a "base class"—another way to think about it is "a generic object class"—that holds attributes found in all objects appearing on the screen.The base class might be called "shape" to identify it in a generic sort of way. You would then "derive" different objects—circles, squares, lines—from this base class, adding in new attributes that are specific to these objects. A specific circle drawn on the screen using the circle class would then be an "instance" of the circle class, which inherited some of its behavior from the more generic shape class.

6.2 Objects

An object is a computer representation of a physical object. Consider the standard English definition of the word object—all nouns can be treated as objects. By this definition, a person is an object, a table is an object, a chair is an object, a room is an object, etc.

"An object is a self-contained entity with well-defined charateristics and behaviour". An object may correspond to a real-world entity. Objects may be concrete or conceptual.

All objects have attributes. A person has arms, legs, a head and a body. A table has a tabletop and legs. A chair could be made of leather. Every object also has uses. A person can walk or run or talk. A table can be used to hold other objects. A chair is used to sit on.

Object-oriented programming is considered superior to standard approaches because we model the real world as closely as possible.

6.2.1 Objects Defined

So what is an object? An object is a component of a program that knows how to perform certain actions and to interact with other pieces of the program. Functions have previously been described as "black boxes" that take an input and spit out an output. Objects can be thought of as "smart" black boxes. That is, objects can know how to do more than one specific task, and they can store their own set of data. Designing a program with objects allows a programmer to model the program after the real world. A program can be broken down into specific parts, and each of these parts can perform fairly simple tasks. When all these simple pieces are meshed together into a program, it can produce a very complicated and useful application.

Let's say that we are writing a text-based medieval video game. Our video game will have two types of characters: the players and the monsters. A player has to know the values of certain attributes: health, strength, and agility. A player must also know what type of weapon and what type of armor they possess. A player must be able to move through a maze, attack a monster, and pick up treasure. So, to design this "player object", we must first separate data that the player object must know from actions that the player must know how to execute. The definition for a player object could be:

Player Object: data:

- health
- strength
- agility
- type of weapon
- type of armor

actions:

- move
- attack monster
- get treasure

END;

6.2.2 Member Data and Member Functions

Data that an object keeps track of is called member data and actions that an object knows how to do are called member functions. Member data is very similar to variables in a regular function in the sense that no other object can get access to that data (unless given permission by the object). Member data keeps its values over the life of an object.

6.2.3 Objects and Instances

There is a very important distinction between an object and an instance of an object. An object is actually a definition, or a template for instances of that object. An instance of an object is an actual thing that can be manipulated. For instance, we could define a Person object, which may include such member data as hair color, eye color, height, weight,etc. An instance of this object

could be "Dave" and Dave has values for hair color, eye color, etc. This allows for multiple instances of an object to be created. Let's go back to the medieval video game example and define the monster object.

Monster Object: data:
health
skin thickness
claws
tail spikes

actions:
move
attack player with claws
attack player with tail
END;

Now, our game could have one instance of a player:

Player Instance #1: data:
health = 16
strength = 12
agility = 14
type of weapon = "mace"
type of armor = "leather"
END;

and our game could have two instances of monsters: a tough one: and a weak one:

Monster Instance #1: data:
health = 21
skin thickness = 20
claws = "sharp"
tail spikes = "razor sharp"
END;

Monster Instance#2: data:
health = 9
skin thickness = 5
claws = "dull"
tail spikes = "quite dull"
END;

Notice how an instance of an object contains information on member data, but holds nothing about member functions. Every instance of the Monster object performs "attack player" the same way. There is a series of steps in this member function. But each instance of the monster has its own value for the member data. In the preceding example, we can tell the two monsters in our game apart, because of their member data. One monster is tough and the other monster is weak. Let's say that we had a "Battle" function in our game. The pseudocode for it may go something like the following:

```
Function Battle(parameters: _player = the Player Object instance
                _monster = the Monster Object
instance)
   turn = PLAYER;
   while ((_player's health > 0) AND (_monster's health > 0)) {
      if (turn == PLAYER){
         player attack's monster;
         turn = MONSTER;
      }
      else {
         monster attack's player
         turn = PLAYER;
      }
   }
} // END FUNCTION Battle
```

In the "attack" phase, the attacking person would somehow deduct points from the defending person. Let's say that the player was fighting with the weaker monster. The weaker monster's health value is 9. If the player attacked the monster and did 5 points of damage to the monster, the monster's new health value would be 4. The monster keeps this value as its health value until it is updated again. So if the monster ran away at this point, and later in the game, the player discovered the weaker monster again, it's health value would still be 4.

6.3 Classes

Every object must have a specific type just like any other data definition. Type definitions for objects are made using the class construct. Classes are not objects just the definition thereof. Objects are created using the standard variable creation procedures — they are then called instances of the class. Classes are an extension of the standard struct feature. Struct allows us to group data items together. Classes allow us to group together data items and functions which operate on the data. This makes the object into a self-contained unit, cutting down on reliance or other functions. This is the first important feature of any object-oriented paradigm.

The general syntax for the class keyword is:

```
class ClassName
   {
protections: data_structures; protection; methods; }
```

Where class is the keyword to start a class definition. Class is the name of the class being defined. protection declares access protection, usually private for data and public for methods. data_structures are any normal C# data structures. Methods are the functions that have access to the data defined in the class.

6.4 Objects as Data types

In OOPs, we say that objects are members of classes. What does this mean? Lets look at an analogy. Almost all computer languages have built-in data types. for instance, a data type int, meaning integer, is pre-defined in C#. You can declare as many variables of type int as you need in your program:

```
int day; int count; int divisor; int answer;
```

In a similar way, you can define many objects of the same class, as shown in class serves as a plan, or blueprint. It specifies what data and what functions will be included in object of that class. Defining the class doesn't create any objects, just as mere existence of a type int does not create any variables.

A class combines data and procedures into a single programming unit. This is encapsulation. Data and procedures are tied together logically. The parts that make up a class can be protected from the rest of the program to varying degrees. The programmer has control over the protection of individual entries in the class definition.

6.5 Creating Classes

Now that we've covered how to create and use objects, and how objects are cleaned up, it's time to show you how to write the classes from which objects are created. This section shows you the main components of a class through a small example that implements a last-in-first-out (LIFO) stack. The following diagram lists the class and identifies the structure of the code.

This implementation of a stack uses another object, a Vector, to store its elements. Vector is a growable array of objects and does a nice job of allocating space for new objects as space is required. The Stack class makes use of this code by using a Vector to store its elements. However, it imposes LIFO restrictions on the Vector— that is, you can only add elements to and remove elements from the top of the stack.

6.5.1 The Class Declaration

The previous diagram shows that two primary components make up the implementation of a class: the class declaration and the class body. The class declaration declares the name of the class along with other attributes. The class declaration for the Stack class is fairly simple and indicates that the class is public and that its name is Stack. Often, a minimal class declaration such as this one is all you will need. However, the class declaration can say more about the class, such as the name of the superclass and if it can be subclassed.

6.5.2 The Class Body

The class body follows the class declaration and is embedded within curly braces { and }. The class body contains declarations for all instance variables and class variables (known collectively as member variables)for the class. In addition, the class body contains declarations and implementations for all instance methods and class methods (known collectively as methods) for the class.

6.5.3 Classes and Members

You make a new type by declaring a class. A class is just a collection of variables-often of different types-combined with a set of related functions.

One way to think about a car as a collection of wheels, doors, seats, windows, and so forth. Another way is to think about what a car do: It can move speed up slow down, stop, park, and so on. A class enables you to encapsulate, or bundle, these various parts and various functions into one collection, which is called an object.

Encapsulating everything you know about a car into one class has a number of advantages for a programmer. Everything is in one place, which makes it easy to refer to copy, and manipulate the data. Likewise, clients of your class—that is the parts of the program that use your class—can use your object wihout worry about is in it or how it works.

A class can consist of any combination of the variable type and also other class types. The variables in the class are referred to as the member variables or data members. A car class might have member variables representing the seats, radio type, tires and so forth.

Member variables also known as data members are the variables in your class. Member variables are part of your class, just like the wheels and engine are part of your car.

The function in the class typically manipulate the member variables. They are referred to as member function or methods of the class. Methoda of the car class might include start () and Brake (). A cat class might have datamembers that represent age and weight; its methods might include sleep (). Memory (), and chasemice().

```
class animal { public:
   int weight;
   int feet;
};
int main() { animal dog1, dog2, chicken; animal cat1; class animal
cat2;
   dog1.weight = 15;
   dog2.weight = 37;
   chicken.weight = 3;
   dog1.feet = 4;
   dog2.feet = 4;
   chicken.feet = 2;
   cout << "The weight of dog1 is " << dog1.weight << "\n";
    cout << "The weight of dog2 is " << dog2.weight << "\n";
```

```
    cout << "The weight of dog2 is " << dog2.weight << "\n";
    cout << "The weight of chicken is " << chicken.weight << "\n";
    return 0; }
// Result of execution
// The weight of dog1 is 15
// The weight of dog2 is 37
// The weight of chicken is 3
```

Examine the example program as an example of a class in C#. This is the first class example, but it will not be the last, since the class is the major reason for using C++/C#/java over some other programming language. You will notice the keyword class used in line 4, in exactly the same way that the keyword struct was used in the last program, and they are in fact very similar constructs. There is a definite difference, as we will see, but for the present time we will be concerned more with their similarities.

The word animal in line 4 is the name of the class, and when we define variables of this type in lines 13 through 15, we can either omit the keyword class or include it if desired as illustrated in line 15. In the last program, we declared 5 variables of a structure type, but in this program we declare 5 objects. They are called objects because they are of a class type. The differences are subtle, and as we proceed through this tutorial, we will see that the class construct is indeed very important and valuable. The class was introduced here only to give you a glimpse of what is to come later in this tutorial.

The class is a type which can be used to define objects in much the same way that a structure is a type that can be used to define variables.Your dog named King is a specific instance of the general class of dogs, and in a similar manner, an object is a specific instance of a class. It would be well to take note of the fact that the class is such a generalized concept that there are libraries of prewritten classes available in the marketplace. You can purchase classes which perform some generalized operations such as managing stacks, queues, or lists, sorting data, managing windows, etc. This is because of the generality and flexibility of the class construct.

The new keyword public in line 6, followed by a colon, is necessary in this case because the variables in a class are defaulted to a private type and we could not access them at all without making them public. Don't worry about this program yet, we will cover all of this in great detail later in this chapter. Be sure to compile and run this example program to see that it does what we say it does with your compiler. Keep in mind that this is your first example of a class and it illustrates essentially nothing concerning the use of this powerful C++ construct.

6.5.4 Scope of Class Members: Public, Protected, and Private

The difference between the public, protected and private members is their scope (which means where they are accessable from). Giving members a scope has two advantages.

1. It tells the compiler how to handle two variables with the same name.
2. It protects the programmer from making mistakes.

The types of protection available within a class are:

- Public
- Private
- Protected

1. Public Public items can be accessed from anywhere in the program without restrictions. Class methods are usually public. As a general rule, data should not be declared public.
2. Private Items within a class marked as private can only be accessed by methods defined as part of the class. Data is most often defined as private.
3. Protected Items declared protected can be referenced directly within the class or in many class that inherits the class.

6.5.5 Using a Class (Instantiation)

Just as writing a class declaration was an extension of concepts we have already covered, the same is true of the syntax needed to instantiate an object. Instantiating an object is what allows you to actually use objects in your program. You can write hundreds of class declarations, but none of that code can be used until you create an instance of an object. A class declaration is merely a template for what an object should look like. When you instantiate an object, C# follows the class declaration as if it was a blueprint for how to create an instance of that object.

6.5.6 Instantiating an Object

You already know how to create an instance of an object. It is exactly the same as using a variable. Let's say you want to have four players in your game, each trying to find their way to the end of a maze faster than the others. You might start by creating four instances of your Player object, like this:

Player blueHat; Player redHat; Player greenHat; Player yellowHat;

It's as easy as that. Although these objects are all Players, they are completely independent of one another.They were created from the same template, but they can have different attributes. For example, "blueHat" might be a slow, strong player, while "greenHat" might be quick and weak, and "yellowHat" might be a well balanced individual, etc. The thing that makes these objects similar is that they all must have values for strength, health, and agility, but nothing else. They can all move, attack monsters, and get treasures, but that is all they can do. So, they are similar in the kinds of things they can do and the attributes they must have, but they are different in that they can each have their own values for those attributes.

6.6 Using an object's member functions

Now that we can create objects, we need to know how to use them. That is, we need to be able to use their member functions. Calling an object's member functions is similar to calling a regular function, with a slight twist..

Suppose someone is playing your adventure game, and they are controlling the player with the green hat. When the person hits the key to attack a monster, you need to make sure that the correct code gets executed so that the right player attacks the monster. The code would look like this:

```
greenHat.attackMonster();
```

It's very similar to the way that a regular function is called, preceeded by the name of the object on which the function should be called (greenHat), and a period. So, instead of calling the function attackMonster(), we are telling greenHat to call the function attackMonster. This is the power of C#. You can very easily "delegate responsibility" to the objects you've written, so what seems like an impossible task of writing a video game, becomes much more manageable. Now let's suppose that greenHat picks up a treasure. Somewhere in your program, the following will get executed:

```
greenHat.getTreasure();
```

This will cause the function getTreasure to be executed for the player with the green hat only. The other players in the game will not execute this function. Perhaps you write getTreasure to look like this:

```
void Player::getTreasure() {
  health++;  // increments the value of the player's health by one
}
```

Suppose that before greenHat picked up the treasure, the following were the players' values for their healths:

blueHat: 14 redHat: 11 yellowHat: 8 greenHat: 11

After greenHat picked up the treasure, the values would be:

blueHat: 14 redHat: 11 yellowHat: 8 greenHat: 12

Notice that only greenHat's health increased when the function Player::getTreasure was called; that's because the member function was called on greenHat only.

6.7 Most simple C# Program with a class

The most simple C# program is shown here:

Listing 6.1

```
// in Simple.cs
class App {
   public static void Main()
   {

   }

}
```

This code does nothing, but will compile using the C# compiler csc.exe:

csc Simple.cs

Before making this application do more let us examine the code that you already seen. The code defines a .NET class which has a single public method. Functions in C# are known as methods. By public I mean that the method can be called from code outside of the class, in general a method marked as private is only accessible from code in the current class. However Main() is a special case, you can declare it as private and the system will still be able to call it, but in practice you should always declare it as public.

6.8 Static Methods and new Operator

The method is also marked as static. Classes are usually used to create objects — instances of the class — and methods are called through objects using data that is specific to an object. However, a static method can be called without an object instance. .NET does not allow you to write global methods, every method has to be part of a class, so static is the only way to call a method without first creating an object.

Because static methods are not associated with objects, they cannot call non-static methods or use other members of the class that is non-static. For example:

Listing 6.2

```
class App {
   public static void Main()
   {
      f(); // this will not compile!
   }
   public void f()
   {
   }
}
```

One typical reason for a static method is to create an instance of the class, this is done in C# with the new operator:

Listing 6.3

```
class App
{
   public static void Main()
   {
      App app = new App();
      app.f();
   }
   public void f()
   {
   }
}
```

The two new lines create an instance of the App class and call the non-static method f() through that instance; app is known as a reference.The syntax used here is different to what you expect in C++. Firstly, the code must explicitly specify the constructor that will be used (in this case the default constructor that is provided by C#). The second difference is that although new is used to create a reference there is no equivalent of the C++ delete operator. The reason is that .NET has a garbage collector that monitors object usage and when all references to an object have been released (or if there is a circular dependency) the garbage collector can release the object. In C++ new has the specific meaning of "create a new instance of this class in the C++ free store". The meaning of new in C# has wider range, it merely says "create a new instance of this class", usually this class will be created on the heap and managed by the .NET garbage collec-

tor, but if it is used to create an instance of a value type (a term that will be explained later) the object will be created on the stack.

> Note: if you have a class that holds resources that should not be held for a long time (for example an exclusive lock on a file), then you should implement a method on the class to free this resource and explicitly call this method when you are sure that the object will not longer be used. Typically, such a method is called Dispose().

The Main() function has a special meaning, it is the entry point for an application, and every C# application must have a class with a public static Main() method — and only one such class. As in C++, the Main() method may return void or an int and it can take parameters. Main() that accepts command line parameters looks like this:

```
public static int Main(string[] args) {
   return 0;
}
```

The args parameter is an array of strings. Arrays in .NET are instances of the System.Array class, and items are accessed through square brackets, as in C++. The size of an array is determined by accessing the Length property. A property gives access to data in the class.

For your application to do anything it must use a class library. A class library contains classes that perform various system actions like creating and manipulating windows, managing security, or accessing databases. C# code uses the .NET library that is common to all code that runs on the .NET platform. This means that the same library is available to VB.NET code and through the Managed Extensions for C++.

6.9 Creating an Object

Listing 6.4

```
//sharp53.cs
class hema {
   static void Main()
   {
     test a;
     a.alpha();
   }
}
class test {
   public void alpha()
   {
     System.Console.WriteLine("alpha");
   }
}
```

The output of this program shall be displayed on console as:

```
C:\>csc c:\sharp\sharp1\sharp53.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp53.cs(7,4): error CS0165: Use of unassigned local variable 'a'

C:\>
```

This program consists of two classes hema and test. Class test has one method (function) alpha(). In class hema we are invoking this method through an assignment a.alpha() where "a" represents class test. This has been done in anology to what we do when we say "int a;" or "string s". However, it is not possible to say "test a". Hence the error message. Let us call "a" an object as it is not a variable.

Listing 6.5

```
//sharp54.cs
class hema {
   static void Main()
   {
     test a;
     a = new test();
     a.alpha();
   }
}
class test {
   public void alpha()
   {
     System.Console.WriteLine("alpha");
   }
}
```

The output of this program shall be displayed on console as:

```
C:\>csc c:\sharp\sharp1\sharp54.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.


C:\>sharp54
alpha

C:\>
```

This program introduces the concept of object and instances of a class. Test is the name of an existing class and we want to make it an object "a". The statement "a = new test();" says to create a new instance of class test() and use it as an object. The word "new" is a C# keyword as in c++.Do not forget to put the brackets () after the class name. We can say that we have an object "a" that looks like the class test. The object is nothing but an instance of a class.

6.9.1 Static Methods

Listing 6.6

```
//sharp55.cs
class hema {
   static void Main()
   {
     test a;
     a=new test();
     a.alpha();
     test.beta();
   }
}
class test {
   public void alpha()
   {
     System.Console.WriteLine("alpha");
   }
   public static void beta()
   {
     System.Console.WriteLine("beta");
   }
}
```

The output of this program shall be displayed on console as:

```
C:\>csc c:\sharp\sharp1\sharp55.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>sharp55
alpha
beta

C:\>
```

This program consists of two methods alpha() and beta().While defining the beta() we have added the word static. All static methods can be called freely. To callor access this static function we have used the statement "test.beta()" and to access the other non-static function alpha() we have used the statement a.alpha(); as an object.If you do the reverse you are likely to get error message as shown at the end of next example.

Listing 6.7

```
//sharp56.cs
class hema {
   static void Main()
   {
     test a;
     a = new test();
```

```
        test.alpha();
        a.beta();
    }
}
class test {
    public void alpha()
    {
        System.Console.WriteLine("alpha");
    }
    public static void beta()
    {
    System.Console.WriteLine("beta");
    }
}
```

The output of this program shall be displayed on console as:

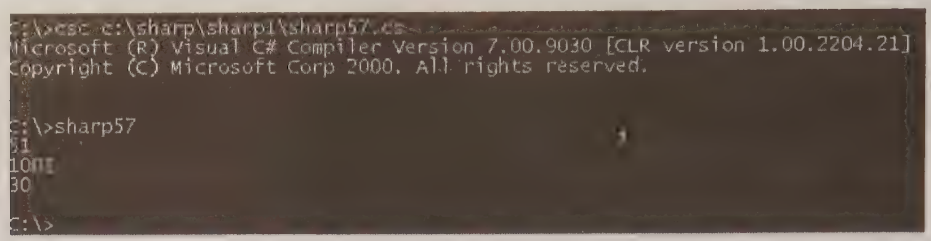
```
C:\>csc c:\sharp\sharp1\sharp56.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp56.cs(8,4): error CS0120: An object reference is required for
the nonstatic field, method, or property 'yyy.alpha()'
sharp\sharp1\sharp56.cs(3,4): error CS0176: Static member 'yyy.beta()' cannot be
accessed with an instance reference; use typename instead

C:\>
```

This example does just the reverse of previous example due to which we get the error message.

Listing 6.8 //sharp57.cs

```
class hema {
    static void Main()
    {
        test a=new test();
        test b=new test();
        a.j=51;
        System.Console.WriteLine(a.j);
        System.Console.WriteLine(b.j);
        test.i=30;
        System.Console.WriteLine(test.i);
    }
}
class test {
    public static int i=101;
    public int j=1001;
}
```

The output of this program shall be displayed on console as:



This example teaches that any static variable can be referenced by prefacing it with the name of the class where it is contained. Let us say in this case static variable "int i" is referenced as test.i in class hema.

6.10 Providing Constructors for Your Classes

A class may contain one or more constructors that provide for the initialization of an object created from the class. This section shows you how to write a constructor. Declaring Member Variables A class's state is represented by its member variables. You declare a class's member variables in the body of the class. Typically, you declare a class's variables before you declare its methods, although this is not required.

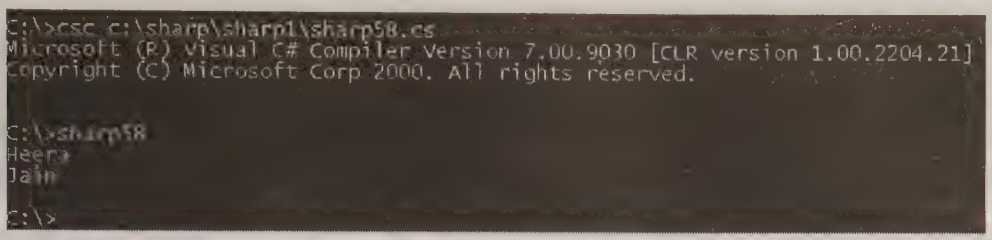
```
classDeclaration {
    member variable declarations
    method declarations
}
```

Note: To declare variables that are members of a class, the declarations must be within the class body, but not within the body of a method. Variables declared within the body of a method are local to that method.

Listing 6.9

```
//sharp58.cs
class hema {
   public static void Main()
   {
     test a;
     System.Console.WriteLine("Heera");
     a = new test();
   }
}
class test            //line 13 {
   public test()      //line 15
   {
     System.Console.WriteLine("Jain");
   }
}
```

The output of this program shall be displayed on console as:



This program consists of a class test at line 13 and function test() at line 15.There are two new things that we are going to learn here:

1. We have assigned the same name to a class and function and there is no harm syntactically
2. The function test is a special function called constructor. A constructor can be used in the cases where every time an object gets created and programmer wants some code to get executed automatically.The code must be putup in the construtor.

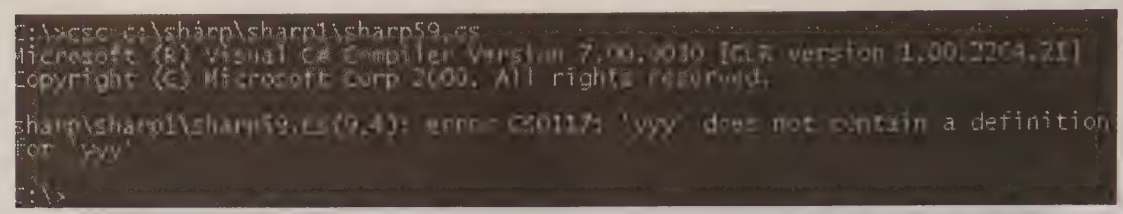
Let us understand what a constructor is:

Observe the output of this program: the word Heera is outputted by virtue of Main function in class hema.and word jain is outputted due to calling of function test() automatically as there is no explicit call of this function anywhere else. The statement test a;specifies that "a" looks like test and however we are not creating an object that looks like test. With the use of keyword new we have created an object a. It is at this instant program calls the constructor i.e. the function test(). Thus as soon as an object is created c# automatically calls the construtor. The construtor has to have the same name as the name of the class.

Listing 6.10

```
//sharp59.cs class xyz59 {
   public static void Main()
{
   yyy a;
   System.Console.WriteLine("Heera");
   a=new yyy();
   a.yyy();
} }
   class yyy
{
   public yyy()
{
   System.Console.WriteLine("Jain");
} }
```

The output of this program shall be displayed on the console as:

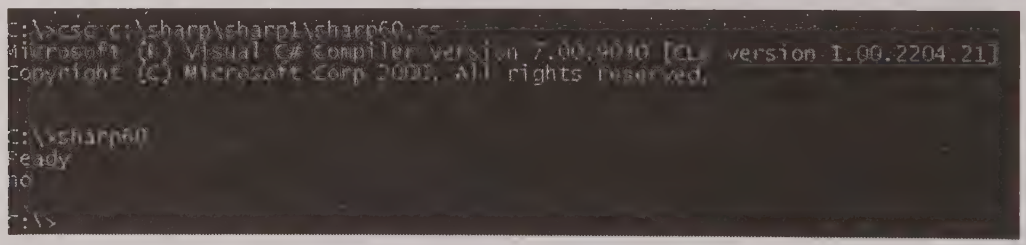

C:\>csc c:\sharp\sharp1\sharp59.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
sharp\sharp1\sharp59.cs(9,4): error CS0117: 'yyy' does not contain a definition
for 'yyy'
C:\>

This program illustrates the concept that when when you have a function with the same name as that of the class, you cannot call the function at all. Due to this, the compiler issues an error message.

Listing 6.11 //sharp60.cs

```
    class hema {
    public static void Main()
    {
      test a;
      System.Console.WriteLine("Ready");
      a = new test("no");
    }
}
class test {
    public test(string s)
    {
      System.Console.WriteLine(s);
    }
}
```

The output of this program shall be displayed on console as:



The difference between this and previous example is that this example shows how the parameters are passed upon.

Listing 6.12

```
//sharp61.cs
class hema {
    public static void Main()
    {
      test alpha,beta;
      System.Console.WriteLine("Ready");
      alpha = new test("no");
```

```
        beta = new test();
    }
}
    class test
    {
      public test(string s)
      {
        System.Console.WriteLine(s);
      }
    public test()
    {
      System.Console.WriteLine("let us go");
    }
}
```

The output of this program shall be displayed on the console as:

This example has two constructors alpha and beta. Alpha has a parameter "no" which is passed upon to string s of class test. First object alpha is created and then object beta.

Listing 6.13

```
//sharp62.cs
class hema {
    public static void Main()
    {
      gamma a=new gamma();
    }
}
    public class gamma
    {
      public gamma()
      {
        System.Console.WriteLine("Constructor");
      }
      ~gamma()
     {
       System.Console.WriteLine("Destructor");
      }
}
```

The output of this program shall be displayed on the console as:

```
C:\>csc c:\sharp\sharp1\sharp62.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.


C:\>sharp62
Constructor

C:\>
```

This example has been included to demonstrate the concept of destructors.A destructor is a function with the same name as that of class concerned but starting with a character ~ (This is pronounced as "tilde"). The destructor function gets called upon by garbage collector classes (not included in this example) as when memory buffer needs the values stored in memory by virtue of various variables to be destroyed safely.

6.11 Class Declaration

Class declarations define new reference types. A class can inherit from another class, and can implement interfaces.

Class members can include constants, fields, methods, properties, indexers, events, operators, constructors, destructors, and nested type declarations. Each member has an associated accessibility, which controls the regions of program text that are able to access the member. There are five possible forms of accessibility. These are summarized in the table 6.1

Table 6.1 Five forms Accessibility

Form	Intuitive meaning
public	Access not limited protected
	Access limited to the containing class or types derived from the containing class
internal	Access limited to this program protected internal Access limited to this program or types derived from the containing class
private	Access limited to the containing type

Since the beginning of this book, you have been using classes. By now, you should have a sense of what a class is for and how to specify one. This lesson will build upon what you already know and introduce the various class members.

Classes are declared by using the keyword "class" followed by the class name and a set of class members surrounded by curly braces. Constructors do not have return values and always have the same name as the class. Listing 7.1 is an example of a class.

Listing 6.14: Example C# Classes: name7.cs

```
// Namespace Declaration
using System;
// helper class class OutputClass {
```

```
        string myString;
        // Constructor
        public OutputClass(string inputString) {
            myString = inputString;
        }
        // Instance Method
        public void printString() {
            Console.WriteLine("{0}", myString);
        }
        // Destructor
        ~OutputClass() {
            // Some resource cleanup routines
        }
    }
    // Program start class class ExampleClass {
        // Main begins program execution.
        public static void Main() {
            // Instance of OutputClass
             OutputClass outCl = new OutputClass("This is printed by the
    programmar."); { }
            // Call Output class' method
            outCl.printString();
        } }
```

The output of this program will be:

```
C:\>name7
This is printed by the programmar.
```

Listing 6.14 shows two classes. The top class, "OutputClass", has a constructor, instance method, and a destructor. It also had a field named "myString". Constructors are used to initialize a class' data members. In this case, the OutputClass constructor accepts a string argument. This string is copied to the class field "myString".

Constructors are not mandatory, as indicated by the implementation of ExampleClass. In this case, a default constructor is provided. A default constructor is simply a constructor with no arguments. However, a constructor with no arguments is not always useful. To make default constructors more useful, you can implement them with initializers. Here is an example:

```
public OutputClass() : this("Default Constructor String") { }
```

Imagine this constructor as included already in class OutputClass from Listing 6.14. This default constructor is followed by an initializer. The colon, ":", marks the beginning of the initializer, followed by the "this" keyword. The "this" keyword refers to this particular object. It effectively makes a call to the constructor of the same object it is defined in. After the "this" keyword is a parameter list with a string. The action taken by the initializer above is to invoke the OutputClass constructor that takes a string type as an argument. The initializer helps you to ensure your class fields are initialized when a class is instantiated.

The example above illustrates how a class can have multiple constructors. The specific constructor called depends on the number of parameters and the type of each parameter.

In C#, there are two types of class members — *instance* and *static*. Instance class members belong to a specific occurrence of a class. Every time you declare an object of a certain class, you create a new instance of that class. The ExampleClass Main() method creates an instance of the OutputClass named "outCl". You can create multiple instances of OutputClass with different names. Each of these instances are separate and stand alone. For example, if you create two OutputClass instances as follows:

```
OutputClass oc1 = new OutputClass("OutputClass1");
OutputClass oc2 = new OutputClass("OutputClass2");
```

You create two separate instances of OutputClass with separate "myString" fields and separate "printString()" methods. On the other hand, if a class member is static, you can access it simply by using the syntax <classname>.<static class member>. The instance names are "oc1" and "oc2".

Suppose OutputClass had the following static method:

```
static void staticPrinter() {
    Console.WriteLine("There is only one of me.");
}
```

Then you could call that function from Main() like this:

```
OutputClass.staticPrinter();
```

You must call static class members through their class name and not their instance name. There is only ever one copy of a static class member.

Another type of constructor is the static constructor. You declare a static constructor by using the keyword "static" just in front of the constructor name. A static constructor is called before an instance of a class is created, before a static member is called, and before the static constructor of a derived class (covered in a later chapter). They are called only once.

OutputClass also has a destructor. Destructors look just like constructors, except they have a tilde, "~", in front of them. Destructors are places where you could put code to release any resources your class was holding during its lifetime. They are normally called when the C# garbage collector decides to clean your object from memory.

6.12 All Types of Class Members

So far, the only class members you've seen are Fields, Methods, Constructors, and Destructors. Here is a complete list of the types of members you can have in your classes:

- Constructors
- Destructors
- Fields
- Methods
- Properties
- Indexers
- Delegates
- Events
- Nested Classes

Listing 6.15

```
using System;
class MyClass {
   public MyClass()
   {
     Console.WriteLine("Constructor");
   }
   public MyClass(int value)
   {  MyField = value;
      Console.WriteLine("Constructor");
   }
   ~MyClass()
    {
      Console.WriteLine("Destructor");
     }
     public const int MyConst = 12;
     public int MyField = 34;
     public void MyMethod()
     {
        Console.WriteLine("MyClass.MyMethod");
      }
     public int MyProperty {
          get
          {
               return MyField;
          }
          set
          {
               MyField = value;
          }
     }
     public int this[int index] {
          get
          {
               return 0;
          }
          set
          {
            Console.WriteLine("this[{0}] = {1}", index, value);
          }
     }
     public event EventHandler MyEvent;
     public static MyClass operator+(MyClass a, MyClass b) {
          return new MyClass(a.MyField + b.MyField);
```

```
        }
        internal class MyNestedClass
        {}
    }
```

The listing 6.15 shows a class that contains each kind of member.

Listing 6.16

```
class Test {
    static void Main() {
        // Constructor usage
        MyClass a = new MyClass();
        MyClass b = new MyClass(123);
        // Constant usage
        Console.WriteLine("MyConst = {0}", MyClass.MyConst);
        // Field usage
        a.MyField++;
        Console.WriteLine("a.MyField = {0}", a.MyField);
        // Method usage
        a.MyMethod();
        // Property usage
        a.MyProperty++;
        Console.WriteLine("a.MyProperty = {0}", a.MyProperty);
        // Indexer usage
        a[3] = a[1] = a[2];
        Console.WriteLine("a[3] = {0}", a[3]);
        // Event usage
        a.MyEvent += new EventHandler(MyHandler);
        // Overloaded operator usage
        MyClass c = a + b;
    }
    static void MyHandler(object sender, EventArgs e) {
        Console.WriteLine("Test.MyHandler");
    }
    internal class MyNestedClass
    {}
}
```

The program in Listing 6.16 shows uses of these members.

6.12.1 Constants

A constant is a class member that represents a constant value which can be computed at compile-time. Constants are permitted to depend on other constants within the same program as long as there are no circular dependencies.

```
class Constants {
     public const int A = 1;
     public const int B = A + 1;
}
```

shows a class named Constants that has two public constants. Even though constants are considered static members, a constant declaration neither requires nor allows a static modifier. Constants can be accessed through the class, as in

```
class Test {
     static void Main() {
     Console.WriteLine("{0}, {1}", Constants.A, Constants.B);
     }
}
```

which prints out the values of Constants.A and Constants.B.

6.12.2 Fields

A field is a member that represents a variable associated with an object or class. The example

```
class Color {
     internal ushort redPart;
     internal ushort bluePart;
     internal ushort greenPart;
     public Color(ushort red, ushort blue, ushort green) {
          redPart = red;
          bluePart = blue;
          greenPart = green;
     }
     -
}
```

shows a Color class that has internal instance fields named redPart, greenPart, and bluePart. Fields can also be static, as shown in the following example:

```
class Color {
     public static Color Red = new Color(0xFF, 0, 0);
     public static Color Blue = new Color(0, 0xFF, 0);
     public static Color Green = new Color(0, 0, 0xFF);
     public static Color White = new Color(0xFF, 0xFF, 0xFF);
     -
}
```

which shows static fields for Red, Blue, Green, and White.

Static fields are not a perfect match for this scenario. The fields are initialized at some point before they are used, but after this initialization there is nothing to stop a client from changing them. Such a modification can cause unpredictable errors in other programs that use Color and assume that the values do not change. Readonly fields can be used to prevent such problems. Assignments to a readonly field can only occur as part of the declaration, or in a constructor in

the same class. Thus, the Color class can be enhanced by adding the readonly modifier for the static fields:

```
class Color {
     internal ushort redPart;
     internal ushort bluePart;
     internal ushort greenPart;
     public Color(ushort red, ushort blue, ushort green) {
          redPart = red;
          bluePart = blue;
          greenPart = green;
     }
     public static readonly Color Red = new Color(0xFF, 0, 0);
     public static readonly Color Blue = new Color(0, 0xFF, 0);
     public static readonly Color Green = new Color(0, 0, 0xFF);
     public static readonly Color White = new Color(0xFF, 0xFF, 0xFF);
}
```

6.12.3 Methods

A method is a member that implements a computation or action that can be performed by an object or class. Methods have a list of formal parameters (which may be empty), a return value (or void), and are either static or non-static. Static methods are accessed through the class. Non-static methods, which are also called instance methods, are accessed through instances of the class.

Listing 6.17

```
using System; public class Stack {
     public static Stack Clone(Stack s) {_}
     public static Stack Flip(Stack s) {_}
     public object Pop() {_}
     public void Push(object o) {_}
     public override string ToString() {_}

} class Test {
     static void Main() {
          Stack s = new Stack();
          for (int i = 1; i < 10; i++)
               s.Push(i);
          Stack flipped = Stack.Flip(s);
          Stack cloned = Stack.Clone(s);
          Console.WriteLine("Original stack: " + s.ToString());
          Console.WriteLine("Flipped stack: " + flipped.ToString());
          Console.WriteLine("Cloned stack: " + cloned.ToString());
     }
}
```

Listing 6.17 shows a stack that has several static methods (Clone and Flip) and several instance methods (Push, Pop, and ToString). Methods can be overloaded, which means that multiple methods may have the same name so long as they have unique signatures. The signature of a method consists of the name of the method and the number, modifiers, and types of its formal parameters. The signature of a method specifically does not include the return type.

Listing 6.18

```
class Test
  { static void F()
     {
       Console.WriteLine("F()");
     }
     static void F(object o)
     {
        Console.WriteLine("F(object)");
     }
     static void F(int value)
     {
        Console.WriteLine("F(int)");
     }
     static void F(int a, int b)
     {
       Console.WriteLine("F(int, int)");
     }
     static void F(int[] values)
     {
         Console.WriteLine("F(int[])");
     }
     static void Main()
     {
          F();
          F(1);
          F((object)1);
          F(1, 2);
          F(new int[] {1, 2, 3});
     }
}
```

Listing 6.18 shows a class with a number of F methods. The output of the program is

```
F() F(int) F(object) F(int, int) F(int[])
```

6.12.4 Properties

A property is a member that provides access to an attribute of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However,

unlike fields, properties do not denote storage locations. Instead, properties have accessors that specify the statements to execute in order to read or write their values. Properties are defined with property declarations. The first part of a property declaration looks quite similar to a field declaration. The second part includes a get accessor and/or a set accessor. In the following example,

```
Caption property. public class Button {
     private string caption;
     public string Caption {
          get {
               return caption;
          }
          set {
               caption = value;
               Repaint();
          }
     }
}
```

The button class defines a Property that can be both read and written, like the Caption property, include both get and set accessors. The get accessor is called when the property's value is read; the set accessor is called when the property's value is written. In the set accessor, the new value for the property is given in an implicit parameter named value.

Declaration of properties is relatively straightforward, but the true value of properties shows itself in use rather than in declaration. The Caption property can be read and written in the same way that fields can be read and written:

Button b = new Button(); b.Caption = "ABC"; // set; causes repaint string s = b.Caption; // get b.Caption += "DEF"; // get & set; causes repaint

6.12.5 Events

An event is a member that enables an object or class to provide notifications. A class defines an event by providing an event declaration, which resembles a field declaration, though with an added event keyword, and an optional set of event accessors. The type of this declaration must be a delegate type.

In the example

```
public delegate void EventHandler(object sender, System.EventArgs e);
public class Button {
     public event EventHandler Click;
     public void Reset() {
          Click = null;
     }
}
```

the Button class defines a Click event of type EventHandler. Inside the Button class, the Click member corresponds exactly to a private field of type EventHandler. However, outside the Button class, the Click member can only be used on the left hand side of the += and -= operators. The += operator adds a handler for the event, and the -= operator removes an handler for the event.

Listing 6.19

```
using System; public class Form1 {
     public Form1() {
  // Add Button1_Click as an event handler for Button1's Click event
          Button6.Click += new EventHandler(Button1_Click);
     }
     Button Button1 = new Button();
     void Button1_Click(object sender, EventArgs e) {
          Console.WriteLine("Button1 was clicked!");
     }
     public void Disconnect() {
          Button6.Click -= new EventHandler(Button1_Click);
     }
}
```

The program in Listing 6.19 shows a Form1 class that adds Button1_Click as an event handler for Button1's Click event. In the Disconnect method, the event handler is removed.

For a simple event declaration such as public event EventHandler Click; the compiler automatically provides the implementation underlying the += and -= operators. An implementer who wants more control can do so by explicitly providing add and remove accessors. The Button class could be rewritten to include add and remove accessors as follows:

```
public class Button {          private EventHandler handler;
          public event EventHandler Click {
               add { handler += value; }
               remove { handler -= value; }
          }
}
```

This change has no effect on client code, but allows the Button class more implementation flexibility. For example, the event handler for Click need not be represented by a field.

6.12.6 Operators

An operator is a member that defines the meaning of an expression which can be applied to instances of the class. There are three kinds of operators that can be defined: unary operators, binary operators, and conversion operators. The following examples defines a Digit type that represents decimal digits-integral values between 0 and 9.

Listing 6.20

```
using System; public struct Digit {
     byte value;
```

```
    public Digit(byte value) {
        if (value < 0 || value > 9) throw new ArgumentException();
this.value = value;
    }
    public Digit(int value): this((byte) value) {}
    public static implicit operator byte(Digit d) {
        return d.value;
    }
    public static explicit operator Digit(byte b) {
        return new Digit(b);
    }
    public static Digit operator+(Digit a, Digit b) {
        return new Digit(a.value + b.value);
    }
    public static Digit operator-(Digit a, Digit b) {
        return new Digit(a.value - b.value);
    }
    public static bool operator==(Digit a, Digit b) {
        return a.value == b.value;
    }
    public static bool operator!=(Digit a, Digit b) {
        return a.value != b.value;
    }
    public override bool Equals(object value) {
        return this == (Digit) value;
    }
    public override int GetHashCode() {
        return value.GetHashCode();
    }
    public override string ToString() {
        return value.ToString();
    }
} class Test {
    static void Main() {
        Digit a = (Digit) 5;
        Digit b = (Digit) 3;
        Digit plus = a + b;
        Digit minus = a - b;
        bool equals = (a == b);
        Console.WriteLine("{0} + {1} = {2}", a, b, plus);
        Console.WriteLine("{0} - {1} = {2}", a, b, minus);
        Console.WriteLine("{0} == {1} = {2}", a, b, equals);
    }
}
```

The Digit type defines the following operators:

- An implicit conversion operator from Digit to Byte.
- An explicit conversion operator from Byte to Digit.
- An addition operator that adds two Digit values and returns a Digit value.
- A subtraction operator that subtracts one Digit value from another, and returns a Digit value.
- The equality (==) and inequality (!=) operators, which compare two Digit values.

6.12.7 Indexers

An indexer is a member that enables an object to be indexed in the same way as an array. Whereas properties enable field-like access, indexers enable array-like access.

As an example, consider the Stack class presented earlier. This class might want to expose array-like access so that it is possible to inspect or alter the items on the stack without performing unnecessary Push and Pop operations. The Stack is implemented as a linked list, but wants to provide the convenience of array access.

Indexer declarations are similar to property declarations, with the differences are that the indexers are nameless (the "name" used in the declaration is this, since this is being indexed) and that indexers include indexing parameters. The indexing parameters are provided between square brackets.

Listing 6.21

```
using System; public class Stack {
     private Node GetNode(int index) {
          Node temp = first;
          while (index > 0) {
               temp = temp.Next;
               index–;
          }
          return temp;
     }
     public object this[int index] {
          get {
               if (!ValidIndex(index))
                    throw new Exception("Index out of range.");
               else
                    return GetNode(index).Value;
          }
          set {
               if (!ValidIndex(index))
                    throw new Exception("Index out of range.");
               else
                    GetNode(index).Value = value;
          }
     }
     –
```

```
}
class Test {
     static void Main() {
          Stack s = new Stack();
          s.Push(1);
          s.Push(2);
          s.Push(3);
          s[0] = 33;      // Changes the top item from 3 to 33
          s[1] = 22;      // Changes the middle item from 2 to 22
          s[2] = 11;      // Changes the bottom item from 1 to 11
     }
}
```

The program in 6.21 shows an indexer for the Stack class.

6.12.8 Instance constructors

An instance constructor is a member that implements the actions required to initialize an instance of a class.

Listing 6.22

```
using System;
     class Point {
     public double x, y;
     public Point() {
          this.x = 0;
          this.y = 0;
     }
     public Point(double x, double y) {
          this.x = x;
          this.y = y;
     }
     public static double Distance(Point a, Point b) {
          double xdiff = a.x - b.x;
          double ydiff = a.y - b.y;
          return Math.Sqrt(xdiff * xdiff + ydiff * ydiff);
     }
     public override string ToString() {
          return string.Format("({0}, {1})", x, y);
     }
} class Test {
     static void Main() {
          Point a = new Point();
          Point b = new Point(3, 4);
          double d = Point.Distance(a, b);
          Console.WriteLine("Distance from {0} to {1} is {2}", a, b,
d);
     }
}
```

The program in Listing 6.22 shows a Point class that provides two public constructors. One Point constructor takes no arguments, and the other takes two double arguments.
If no constructor is supplied for a class, then an empty constructor with no parameters is automatically provided.

6.12.9 Destructors

A destructor is a member that implements the actions required to destruct an instance of a class. Destructors cannot take parameters, cannot have accessibility modifiers, and cannot be called explicitly. The destructor for an instance is called automatically during garbage collection.

Listing 6.23

```
using System;
     class Point {
     public double x, y;
     public Point(double x, double y) {
          this.x = x;
          this.y = y;
     }
     ~Point() {
          Console.WriteLine("Destructed {0}", this);
     }
     public override string ToString() {
          return string.Format("({0}, {1})", x, y);
     }
}
```

Listing 6.23 shows a Point class with a destructor.

6.12.10 Static constructors

A static constructor is a member that implements the actions required to initialize a class. Static constructors cannot take parameters, cannot have accessibility modifiers, and cannot be called explicitly. The static constructor for a class is called automatically when the class is loaded.

Listing 6.24

```
using System.Data; class Employee {
     private static DataSet ds;
     static Employee() {
          ds = new DataSet(_);
     }
     public string Name;
     public decimal Salary;
     _
}
```

The program in Listing 6.24 shows an Employee class with a static constructor that initializes a static field.

6.13 Inheritance

Classes support single inheritance, and the type object is the ultimate base class for all classes. All the classes shown in earlier examples are implicitly derived from object.

```
class A {
     public void F() { Console.WriteLine("A.F"); }
} shows a class A that implicitly derives from object. The example
class B: A {
     public void G() { Console.WriteLine("B.G"); }
} class Test {
     static void Main() {
          B b = new B();
          b.F();                    // Inherited from A
          b.G();                    // Introduced in B

          A a = b;                  // Treat a B as an A
          a.F();
     }
}
```

shows a class B that derives from class A. The class B inherits class A's F method, and introduces a method called G method. Methods, properties, and indexers can be virtual, which means that their implementation can be overridden in derived classes.

Listing 6.25

```
using System;
   class A {
     public virtual void F() { Console.WriteLine("A.F"); }
} class B: A {
     public override void F() {
          base.F();
          Console.WriteLine("B.F");
     }
} class Test {
     static void Main() {
          B b = new B();
          b.F();
          A a = b;
          a.F();
```

Listing 6.25 shows a class A with a virtual method F, and a class B that overrides F. The overriding method in B contains a call base.F() which calls the overridden method in A. A class can indicate that it is incomplete, and is intended only as a base class for other classes by including the abstract modifier. Such a class is called an abstract class. An abstract class can specify abstract members which a non-abstract derived class must implement.

Listing 6.26

```
using System;
   abstract class A {
     public abstract F();
} class B: A {
     public override F() { Console.WriteLine("B.F"); }
} class Test {
     static void Main() {
          B b = new B();
          B.F();
          A a = b;
          a.F();
     }
}
```

Listing 6.26 introduces an abstract method F in the abstract class A. The non-abstract class B provides an implementation for this method.

6.14 Controlling Access to Members of a Class

Member variables and methods are known collectively as members. When you declare a member of a C# class, you can allow or disallow other objects of other types access to that member through the use of access specifiers.

One of the benefits of classes is that classes can protect their member variables and methods from access by other objects. Why is this important? Well, consider this. You're writing a class that represents a query on a database that contains all kinds of secret information, e.g. employee records or income statements of your startup company. Certain information and queries contained in the class, the ones supported by the publicly accessible methods and variables in your query object, are OK for the consumption of any other object in the system. Other queries contained in the class are there simply for the personal use of the class. They support the operation of the class but should not be used by objects of another type—you've got secret information to protect. You'd like to be able to protect these personal variables and methods at the language level and disallow access by objects of another type.

In C#, you can use access specifiers to protect both a class's variables and its methods when you declare them. Like Java C# supports four distinct access levels for member variables and methods: private, protected, public, and, if left unspecified, package.

The following chart shows the access level permitted by each specifier.

Specifier	class	subclass		package		world
private	X	protected		X	X*	X
public	X	X	X	X		
package	X	X				

The first column indicates whether the class itself has access to the member defined by the access specifier or not. As you can see, a class always has access to its own members. The second column indicates whether subclasses of the class (regardless of which package they are in) have access to the member. The third column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The fourth column indicates whether all classes have access to the member.

Note that the protected/subclass intersection has an '*' This particular access case has a special caveat discussed in detail later.

Let us now look at each access level in more detail.

6.14.1 Private

The most restrictive access level is private. A private member is accessible only to the class in which it is defined. You can use this access to declare members that can only be used by the class. This includes variables that contain information that if accessed by an outsider could put the object in an inconsistent state, or methods that, if invoked by an outsider, could jeopardize the state of the object or the program in which it's running. Private members are like secrets we do not tell anybody. To declare a private member, use the private keyword in its declaration. The following class contains one private member variable and one private method:

```
class Alpha {
    private int iamprivate;
    private void privateMethod()
    {
        System.out.println("privateMethod");
    }
}
```

Objects of type Alpha can inspect or modify the **iamprivate** variable and can invoke privateMethod, but objects of other types cannot. For example, the Beta class defined here:

```
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprivate = 10;      // illegal
        a.privateMethod();      // illegal
    }
}
```

cannot access the iamprivate variable or invoke privateMethod on an object of type Alpha because Beta is not of type Alpha. When one of your classes is attempting to access a member varible to which it does not have access, the compiler prints an error message and refuses to compile your program.

If your program is attempting to access a method to which it does not have access, you will see a compile time error.

You may ask how can one Alpha object can access the private members of another Alpha object.This is illustrated by the following example. Suppose the Alpha class contained an instance method that compares the current Alpha object (this) to another object based on their iamprivate variables:

```
class Alpha {
    private int iamprivate;
    bool isEqualTo(Alpha anotherAlpha) {
        if (this.iamprivate == anotherAlpha.iamprivate)
            return true;
        else
            return false;
    }
}
```

This is perfectly legal. Objects of the same type have access to one another's private members. This is because access restrictions apply at the class or type level (all instances of a class) rather than at the object level (this particular instance of a class).

Note: this is a Java language keyword that refers to the current object.

6.14.2 Protected

The next access level specifier is protected, which allows the class itself, subclasses (with the caveat that we referred to earlier), and all classes in the same package to access the members. Use the protected access level when it's appropriate for subclasses of a class to have access to the member, but not unrelated classes. Protected members are like family secrets—you do not mind if the whole family knows, and even a few trusted friends but you would not want any outsiders to know. To declare a protected member, use the keyword protected.Let's look at how the protected specifier affects access for classes in the same package. Consider this version of the Alpha class which is now declared to be within a package named Greek and which has one protected member variable and one protected method declared in it:

```
package Greek;
public class Alpha {
    protected int iamprotected;
    protected void protectedMethod() {
        System.out.println("protectedMethod");
    }
}
```

Now, suppose that the class Gamma was also declared to be a member of the Greek package (and is not a subclass of Alpha). The Gamma class can legally access an Alpha object's member variable and can legally invoke its protectedMethod:

```
package Greek;
class Gamma {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprotected = 10;     // legal
        a.protectedMethod();     // legal
    }
}
```

Let us now investigate as to how the protected specifier affects access for subclasses of Alpha. Let's introduce a new class, Delta, that derives from Alpha but lives in a different package—Latin. The Delta class can access both iamprotected and protectedMethod, but only on objects of type Delta or its subclasses. The Delta class cannot access iamprotected or protectedMethod on objects of type Alpha. accessMethod in the following code sample attempts to access the iamprotected member variable on an object of type Alpha, which is illegal, and on an object of type Delta, which is legal.Similarly, accessMethod attempts to invoke an Alpha object's protectedMethod which is also illegal:

```
package Latin;
import Greek.*;
class Delta extends Alpha {
    void accessMethod(Alpha a, Delta d) {
        a.iamprotected = 10;     // illegal
        d.iamprotected = 10;     // legal
        a.protectedMethod();     // illegal
        d.protectedMethod();     // legal
    }
}
```

If a class is both a subclass of and in the same package as the class with the protected member, then the class has access to the protected member.

6.14.3 Public

The easiest access specifier is public. Any class, in any package, has access to its public members. Declare public members only if such access cannot produce undesirable results or if an outsider uses them. There are no personal or family secrets here; this is the stuff you don't mind if anybody knows. To declare a public member, use the keyword public. For example,

```
package Greek;
public class Alpha {
    public int iampublic;
    public void publicMethod() {
        System.out.println("publicMethod");
    }
}
```

Let us rewrite our Beta class one more time and put it in a different package than Alpha and make sure that it is completely unrelated to (not a subclass of) Alpha: package Roman;

```
import Greek.*;
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampublic = 10;        // legal
        a.publicMethod();        // legal
    }
}
```

As you can see from the above code snippet, Beta can legally inspect and modify the iampublic variable in the Alpha class and can legally invoke publicMethod.

6.14.4 Package

The package access level is what you get if you don't explicitly set a member's access to one of the other levels. This access level allows classes in the same package as your class to access the members. This level of access assumes that classes in the same package are trusted friends. This level of trust is like that which you extend to your closest friends but wouldn't trust even to your family. For example, this version of the Alpha class declares a single package-access member variable and a single package-access method. Alpha lives in the Greek package:

```
package Greek;
class Alpha {
    int iampackage;
    void packageMethod() {
        System.out.println("packageMethod");
    }
}
```

The Alpha class has access both to iampackage and packageMethod. In addition, all the classes declared within the same package as Alpha also have access to iampackage and packageMethod. Suppose that both Alpha and Beta were declared as part of the Greek package:

```
package Greek;
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampackage = 10;      // legal
        a.packageMethod();      // legal
    }
}
```

Beta can legally access iampackage and packageMethod.

6.15 Cleaning Up Unused Objects

Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error prone. The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them. The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called garbage collection. An object is eligible for garbage collection when there are no more references to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value null. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

6.16 The Garbage Collector

The C# runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically, although, in some situations, you may want to run the garbage collection explicitly by calling the gc method in the System class. For instance, you might want to run the garbage collector after a section of code that creates a large amount of garbage or before a section of code that needs a lot of memory.

6.17 Finalization

Before an object gets garbage-collected, the garbage collector gives the object an opportunity to clean up after a call to the object's finalize method. This process is known as *finalization*. Most programmers don't have to worry about implementing the finalize method. In rare cases, however, a programmer might have to implement a finalize method to release resources, such as native peers, that aren't under the control of the garbage collector.

The finalize method is a member of the Object class, which is the top of the class hierarchy and a superclass of all classes. A class can override the finalize method to perform any finalization necessary for objects of that type. We shall learn about this in other chapters.

6.18 Implementing Classes

In order to implement a class, you may need to undertake the following two things:

1. Declare the interface to the class
2. Define the implementation of the class.

We will look at each of these in turn.

In COM+, all classes exist within a single hierarchical namespace. In C#,the using statement lets you avoid to specify the fully qualified name when you use a class. For example, the System namespace contains several classes, including Console. Console has a WriteLine method that, as you might expect, writes a line to the system console. If you want to write the output part of a Hello World program in C#, you can say:

```
System.Console.WriteLine("Hello World!");
```

This same code can be written as:

```
Using System; Console.WriteLine("Hello World!");
```

That's almost everything you need for the C# Hello World program. A complete C# program needs a class definition and a Main function. A complete, console-based Hello World program in C# looks like this:

```
using System; class HelloWorld {
    public static int Main(String[] args)
    {
        Console.WriteLine("Hello, World!");
        return 0;
    }
}
```

The first line makes System—the COM+ base class namespace—available to the program. The program class itself is named HelloWorld (code is arranged into classes, not by files). The Main method (which takes arguments) is defined within HelloWorld. The COM+ Console class writes the friendly message, and the program is finished. But if we want to reuse the HelloWorld program, we put it into its own namespace! For this, just wrap it in a namespace and declare the classes as public if you want them accessible outside the particular namespace. (Here you may note that we have changed the name Main to the more suitable name SayHi.)

```
using System; namespace MSDNMag {  public class HelloWorld
    {  public static int SayHi()
        { Console.WriteLine("Hello, World!");
            return 0;
        }
    }
}
```

You can then compile this into a DLL, and include the DLL with any other programs you are building. The calling program could look like this:

```
using System; using MSDNMag;
class CallingMSDNMag {    public static void Main(string[] args)
    {
        HelloWorld.SayHi();
        return 0;
    }
}
```

One final point about classes. If we have classes with the same name in more than one namespace, C# allows us to define aliases for any of them so that we don't have to fully qualify them. Here is an example. Suppose you have created a class NS1.NS2. ClassA looks like this:

```
namespace NS1.NS2 {
    class ClassA {}
}
```

You can then create a second namespace, NS3, that derives the class N3.ClassB from NS1.NS2.ClassA like this:

```
namespace NS3 {
    class ClassB: NS1.NS2.ClassA {}
}
```

If this construct is too long for you, or if we are going to repeat it several times, we can use the alias A for the class NS1.NS2.ClassA with the using statement like:

```
namespace NS3 {    using A = NS1.NS2.ClassA;
    class ClassB: A {}
}
```

This effect can be accomplished at any level of an object hierarchy. For instance, we could also create an alias for NS1.NS2 like this:

```
namespace NS3 {
    using C = NS1.NS2;
    class ClassB: C.A {}
}
```

Remember the MSDNMag namespace? A single conditional statement can make the SayHi member a debug-only function.

```
using System;
namespace MSDNMag {
    public class HelloWorld
    {
        [conditional("DEBUG")]
        public static void SayHi()
        {
            Console.WriteLine("Hello, World!");
            return;
        }
    ùùù
    }
}
```

Conditional functions must have void return types (as I've set in this sample). The client program would then have to look like this to get a Hello World message:

```
using System using MSDNMag
#define DEBUG
class CallingMSDNMag {
    public static void Main(string[] args)
    {
        HelloWorld.SayHi();
        return 0;
    }
}
```

The code is nice and uncluttered without all those #ifdefs hanging around, waiting to be ignored.

1. What pieces of data (information) must an object of this class remember?
2. Which of these pieces of data might a user wish to access?
3. What operations might the user wish to perform on this object?
4. What classes will be needed as suppliers in order to perform these operations?
5. What additional operations will be required to make the specified operations (#3) available to the user?

If you do not wish to see the completed design at this time, each piece of the design will be introduced before the interface for it is given.

After you have answered all of these questions you are ready to begin writing the interface to your class.

6.19 Class Library and Name Space

A class library is accessed through the using keyword:

```
// in Simple.cs using System;
class App {
   public static int Main(string[] args)
   {
      for(int x = 0; x < args.Length; x++)
         Console.WriteLine(args[x]);
      return 0;
   }
}
```

Here, the using line indicates that the code will use classes defined in the System namespace. A namespace is a collection of .NET classes, which are usually related. As you get more proficient with C#, you will create your own namespaces and putting classes in a namespace allows other applications to use your classes. This code uses a class called Console. As the name suggests this gives access to the command line console. This simple example prints all the command line arguments to the console. Namespaces scope class names by using System in System.String means that you want to use String from the System namespace as opposed to String defined in another namespace. You use the /reference switch on csc to indicate the namespaces that your code will use. The using keyword allows you to use classes without using the fully qualified name.

Note: like C++ and C, each statement must end in a semicolon. However, unlike C++ the class declaration does not have to have a terminating semicolon.

7

Methods

This lesson introduces you to C# Methods. Our objectives in this chapter are as follows:

- Understand the structure of a method.
- Know the difference between static and instance methods.
- Learn to instantiate objects.
- Learn how to call methods of an instantiated object.
- Understand the 4 types of parameters.
- Learn how to use the "this" reference.

7.1 Methods

A method is a member that implements computation or action that can be performed by an object or class. Methods have a list of formal parameters (which may be empty), a return value (or void), and are either static or non-static. Static methods are accessed through the class. Non-static methods, which are also called instance methods, are accessed through instances of the class.

Listing 7.1: Methods (Functions)

```
//sharp36.cs
class test
  {
   static void Main()
   {
     gamma(10,false,"C-Sharp");                    //line 7
   }
   static void gamma(int alpha, bool beta, string s)      //line 9
   {
     System.The console.WriteLine("{0}{1}{2}", alpha, beta, s);
   }
}
```

The output of this program shall be displayed on the console as:

```
C:\>csc c:\sharp\sharp1\sharp36.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>sharp36
10FalseC-Sharp

C:\>
```

Line 9 provides a function definition for function gamma. Since this particular function is to be designed for dealing three different data types, its parameters are defined accordingly as int alpha, bool beta and string s. The function gamma has been called and initialized in line 7 and the these initial parameter has been passed on to function gamma through WriteLine statement.

Listing 7.2

```
//sharp37.cs
 class xyz37
  {
   static void Main()
   {
    gamma(10,false);
   }
   static void gamma(int alpha, bool beta, string s)
   {
   System.The console.WriteLine("{0}{1}{2}",alpha,beta,s);
   }
}
```

The output of this program shall be displayed on the console as:

```
C:\>csc \sharp37.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp37.cs(6,4): error CS1501: No overload for method 'gamma' takes '2' argument
s
sharp37.cs(8,16): (Location of symbol related to previous error)
```

In this program since the function gamma has been called upon in function main with only two parameters the compiler has given the error message as illustrated above.

Listing 7.3

```
//sharp38.cs
  class xyz38 {
   static void Main()
{
   alpha("C-Sharp",10,false);
}
```

```
    static void alpha(int v, bool k, string s)
{
    System.The console.WriteLine("{0}{1}{2}",v,k,s);
} }
```

The output of this program shall be displayed on the the console as:

```
C:\>csc c:\sharp\sharp1\sharp38.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp38.cs(6,4): error CS1502: The best overloaded method match for
 'xyz38.alpha(int, bool, string)' has some invalid arguments
sharp\sharp1\sharp38.cs(6,10): error CS1503: Argument '1': cannot convert from '
string' to 'int'
sharp\sharp1\sharp38.cs(6,20): error CS1503: Argument '2': cannot convert from '
int' to 'bool'
sharp\sharp1\sharp38.cs(6,23): error CS1503: Argument '3': cannot convert from '
bool' to 'string'

C:\>_
```

The error message is due to the mistake made by us in sequencing the parameters in function call and their types in the function definition.

Listing 7.4

```
//sharp39.cs
  class xyz39 {
   static void Main()
   {
      alpha();
    }
}

   class vkj
    {
      static void alpha()
      {
        System.The console.WriteLine("alpha");
     }
   }
```

The output of this program shall be displayed on the the console as:

```
C:\>csc \sharp39.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp39.cs(6,4): error CS0103: The name 'alpha' does not exist in the class or n
amespace 'xyz39'
```

This program consists of two classes—class xyz39 and class vkj. The class xyz39 contains the main() function. As you are already aware, this function is a must for all functions and is a starting point for all programs. We are calling a function alpha() from the class xyz39 whereas the function alpha() resides in class vkj. Therefore the error message.

The output of this program shall be displayed on the console as:

```
C:\>csc c:\sharp\sharp1\sharp40.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\sharp\sharp1\sharp40.cs(6,4): error CS0122: 'yyy.alpha()' is inaccessible due to
 its protection level

C:\>
```

This program uses a concept of shorthand for calling a function of some other class by using the notation vkj.alpha(). In this notation vkj means the class vkj and .alpha() means the function alpha() of class vkj.

In this program everything is alright even then we are getting error message. Why? This is due the fact that function of a class cannot be fetched from another class unless there is permission in the class from where the function is being fetched. This permission is granted by declaring the corresponding function as "public". So this change is made in next program.

Listing 7.5

```
//sharp41.cs
  class xyz40 {
   static void Main()
   {
      vkj.alpha();
   }
}
   class vkj
   {
     public static void alpha()
     {
       System.The console.WriteLine("alpha");
     }
    }
```

The output of this program shall be displayed on the console as:

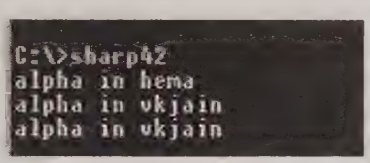
```
C:\>csc c:\sharp\sharp1\sharp41.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.


C:\>sharp41
alpha

C:\>
```

Listing 7.6

```
//sharp42.cs
class vkjain
  {
     static void Main()
  {
```

```
        hema.alpha();
        alpha();
        vkjain.alpha();
       }
       public static void alpha()
{
       System.The console.WriteLine("alpha in vkjain");
}
}
       class hema
       {
        public static void alpha()
        {
          System.The console.WriteLine("alpha in hema");
        }
}
```

The output of this program shall be displayed on the console as:



This example clearly shows the concept of shorthand call of functions. hema.alpha() means use function alpha() of class hema and vkjain.alpha() means function alpha() of class vkjain. Simply calling alpha() in this particular case means vkjain.alpha() by default because we have written this as a part of vkjain class.

Listing 7.7: Passing Parameters by Value

```
//input3.cs
using System;
public class SquareSample {
    public int CalcSquare(int nSideLength)
    {
          return nSideLength*nSideLength;
     }
}
class SquareApp {
    public static void Main()
    {
         SquareSample sq = new SquareSample();
         The console.WriteLine(sq.CalcSquare(25).ToString());
     }
}
```

The output of this program will be:

```
C:\>input3
625
```

Previously, all of our functionality for each program resided in the Main() method. While this was adequate for the simple programs we used to learn earlier concepts, there is a better way to organize your program, using methods. Methods are extremely useful because they allow you to separate your logic into different units.

7.2 Structure of a Method

The structure of a method is as follows:

```
attributes modifiers return-type method-name
  ( parameters )
  { statements }
```

We defer the discussion of attributes and modifiers to a later lesson. The return-type can be any C# type. It can be assigned to a variable to use later in the program. The method name is a unique identifier for what you wish to call a method. To promote understanding of your code, a method name should be meaningful and associated with the task the method performs. Parameters allow you to pass information to and from a method. They are surrounded by parenthesis. Statements within the curly braces carry out the functionality of the method.

Listing 7.8: One Simple Method

```
// One Simple Method:  Method1.cs
  using System;
  class Methodsample {
    public static void Main() {
        string myChoice;
        Methodsample om = new Methodsample();
        do {
            myChoice = om.getChoice();
            // Make a decision based on the user's choice
            switch(myChoice) {
                case "A":
                case "a":
   The console.WriteLine(" Whether you want to add an address.");
                    break;
                case "D":
                case "d":
   The console.WriteLine(" Whether you want to delete an address.");
                    break;
                case "M":
                case "m":
   The console.WriteLine("Whether you wish to modify an address.");
```

```
                    break;
                case "V":
                case "v":
 The console.WriteLine("You wish to view the address list.");
                    break;
                case "Q":
                case "q":
 The console.WriteLine("Thank You.");
                    break;
                default:
 The console.WriteLine("{0} is not a valid choice", myChoice);
            }
            // Pause the screen to allow the user to see the results
            The console.Write("Press any key to continue...");
            The console.ReadLine();
            The console.WriteLine();
        } while (myChoice != "Q" && myChoice != "q");
    // Keep going until the user wants to quit
    }
    string getChoice() {
        string myChoice;
        // Printing  A Menu
        The console.WriteLine("My Address Book\n");
        The console.WriteLine("A - Add New ");
        The console.WriteLine("D - Delete ");
        The console.WriteLine("M - Modify ");
        The console.WriteLine("V - View ");
        The console.WriteLine("Q - Quit\n");
        The console.WriteLine("Choice (A,D,M,V,or Q): ");
        // Retrieving the user's option
        myChoice = The console.ReadLine();
        return myChoice;
    }
}
```

The interactive output of this program shall be:

```
C:\>csc c:\temp\vkj\method1.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
```

Instead of printing the menu and accepting input in the Main() method, this functionality has been moved to a new method called getChoice(). The return type is a string. This string is used in the switch statement in main. The method name "getChoice" describes what happens when it is invoked. Since the parenthesis are empty, no information will be transferred to or from the getChoice() method.

Within the method block we first declare the variable "myChoice". Although this is the same name and type as the "myChoice" variable in Main(), they are both unique variables. They are

local variables and they are visible only in the block they are declared. In other words, the "myChoice" in getChoice() knows nothing about the existence of the "myChoice" in Main(), and visa-versa.

The getChoice() method prints a menu to the the console and gets the user's input. The "return" statement sends the data from the "myChoice" variable back to the caller, Main(), of getChoice(). Notice that the type returned by the "return" statement must be the same as the return-type in the function declaration. In this case it is a string.

In the Main() method, we must instantiate a new "OneMethod" object before we can use getChoice() because of the way getChoice() is declared. Since we did not specify a "static" modifier (as for Main()), getChoice() becomes an instance method.The difference between instance methods and static methods is that multiple instances of a class can be created (or instantiated) and each instance has it's own separate getChoice() method. However, when a method is static, there are no instances of that method, and you can invoke only one definition of the static method.

So, as stated, getChoice() is not static and therefore, we must instantiate a new object to use it. This is done with the declaration "OneMethod om = new OneMethod()". On the left hand side of the declaration is the object reference "om" which is of type OneMethod. The distinction of "om" being a reference is important. It is not an object itself, but it is a variable that can refer (or point) to an object of type OneMethod. On the right hand side of the declaration is an assignment of a new OneMethod object to the reference "om". The keyword "new" is a C# operator that creates a new instance of an object on the heap. What is happening here is that a new OneMethod instance is being created on the heap and then being assigned to the om reference. Now we have an instance of the OneMethod object referenced by om, we can manipulate that instance through the om reference.

Methods, fields, and other class members can be accessed, identified, or manipulated through the "." (dot) operator. Since we want to call getChoice(), we do so by using the dot operator through the om reference: "om.getChoice()". The program then executes the statements in the getChoice() block and returns. To capture the value getChoice() returns, we use the "=" (assignment) operator. The returned string is placed into Main()'s local myChoice variable. From there, the rest of the program executes as you expect, using concepts from earlier lessons.

Listing 7.9

```
//param.cs
  using System;
  class Address {
    public string name;
    public string address;
}
class MethodParams {
    public static void Main() {
        string myChoice;
        MethodParams mp = new MethodParams();
        do {
            // show menu and get input from user
            myChoice = mp.getChoice();
```

```
            // Make a decision based on the user's choice
            mp.makeDecision(myChoice);
            // Pause to allow the user to see the results
            The console.Write("Press any key to continue...");
            The console.ReadLine();
            The console.WriteLine();
        } while (myChoice != "Q" && myChoice != "q");
// Keep going until the user wants to quit
    }
    // show menu and get user's choice
    string getChoice() {
        string myChoice;
        // Print A Menu
        The console.WriteLine("Example Address Book\n");
        The console.WriteLine("A - Add New ");
        The console.WriteLine("D - Delete ");
        The console.WriteLine("M - Modify ");
        The console.WriteLine("V - View ");
        The console.WriteLine("Q - Quit\n");
        The console.WriteLine("Choice (A,D,M,V,or Q): ");
        // Retrieve the user's choice
        myChoice = The console.ReadLine();
        return myChoice;
    }

    // make decision
    void makeDecision(string myChoice) {
        Address addr = new Address();
        switch(myChoice) {
            case "A":
            case "a":
                addr.name = "V.K. Jain";
                addr.address = "11 LF TANSEN MARG";
                this.addAddress(ref addr);
                break;
            case "D":
            case "d":
                addr.name = "Kailash";
                this.deleteAddress(addr.name);
                break;
            case "M":
            case "m":
                addr.name = "Mahesh";
```

```
                this.modifyAddress(out addr);
                The console.WriteLine("Name is now {0}.", addr.name);
                break;
            case "V":
            case "v":
                this.viewAddresses("Hema", "V.K. Jain",
"Mahesh","Kailash");
                break;
            case "Q":
            case "q":
                The console.WriteLine("Thank You.");
                break;
            default:
      The console.WriteLine("{0} is not a valid choice", myChoice);
        }
    }
    // insert an address
    void addAddress(ref Address addr) {
  The console.WriteLine("Name: {0}, Address: {1} added.", addr.name,
addr.address);
    }
    // remove an address
    void deleteAddress(string name) {
  The console.WriteLine("You wish to delete {0}'s address.", name);
    }
    // change an address
    void modifyAddress(out Address addr) {
  //The console.WriteLine("Name: {0}.", addr.name); // causes error!
        addr = new Address();
        addr.name = "V.K. Jain";
        addr.address = "C# Station";
    }
    // show addresses
    void viewAddresses(params string[] names) {
        foreach (string name in names) {
            The console.WriteLine("Name: {0}", name);
        }
    }
}
```

The interactive output of this program may be similar to the following depending on options selected by the user:

Listing 7.9 is a modification of Listing 7.8. There are 4 kinds of parameters a C# method can handle: out, ref, params, and value. To help illustrate usage of parameters, we created an Address class with two string fields.

In Main() we call getChoice() to get the user's input and put that string in the myChoice variable. Then we use myChoice as an argument to makeDecision(). In the declaration of makeDecision() you will notice its one parameter is declared as a string with the name myChoice. Again, this is a new myChoice, separate from the caller's argument and local only to this method. Since makeDecision()'s myChoice parameter does not have any other modifiers, it is considered a "value" parameter.The actual value of the argument is copied on the stack. Variables given by the value parameters are local and any changes to that local variable do not affect the value of the variable used in the caller's argument. In other words, value parameters are input only.

The switch statement in makeDecision() calls a method for each case. These method calls are different than the ones we used in Main(). Instead of using the "mp" reference, they use the "this" keyword. "this" is a reference to the current object. We know the current object has been instantiated because makeDecision() is not a static method. Therefore, we can use the "this" reference to call methods within the same instance.

The addAddress() method takes a "ref" parameter. This means the parameter is passed in as a reference. The reference is copied on the stack when it is passed to the method. This reference still refers to the same object on the heap as the original reference used in the caller's argument. This means any changes to the local reference's object also changes the caller reference's object. You can think of this as a way to have an input/output parameter.

modifyAddress() has an "out" parameter. "out" parameters are only passed back to the calling function. When the method is called, there is only an unassigned reference placed on the stack. Because of definite assignment rules, you cannot use this variable until it has a valid value assigned. The first line in modifyAddress() is commented out on purpose to illustrate this point. Uncomment it and compile to see what happens. Once assigned, the program returns, the value of the "out" parameter will be copied into the caller's argument variable. You must assign a value to an "out" variable before your method returns.

A very useful addition to the C# language is the "params" parameter. This must be a single dimension or jagged array. In makeDecision() we pass in four comma delimited string arguments. The number of arguments is variable. In viewAddresses() we use a foreach loop to print each of these strings. The "params" parameter is considered an input only parameter and any changes affect the local copy only.

In the summary, you understand the structure of a method. You know the four kinds of parameters a method can declare and their proper usage. When you wish to use an instance method, you must instantiate its object as opposed to static methods that can be called any time. You've also learned how the "this" reference is used to call instance methods.

7.3 Method Overloading

C# allows methods to be overloaded. What this means is that a class can have methods with the same name, but that take different parameters. For example, you can write the following for Main():

```
public static void Main()
{
   int i = 1;
   double f = 2.0;
   The console.WriteLine("Integer");
   The console.WriteLine(i);
   The console.WriteLine("Floating point");
   The console.WriteLine(f);
}
```

Other than literal strings, this code passes an integer and a double precision floating point number to WriteThe console(). The output of this new program is:

```
Integer 1 Floating point 2
```

7.4 Implementing Methods

As you know, objects have behavior that is implemented by its methods. Other objects can ask an object to do something by invoking its methods. This section tells you everything you need to know about writing methods for your C# classes. In C#, you define a class's methods in the body of the class for which the method implements some behavior. Typically, you declare a class's methods after its variables in the class body although this is not required.

7.5 Class Containing Functions

Making four function calculator

This section teaches you how to create your own class with some function in it and how to call these functions from main. Here is attached second.cs file.

1. **Creating your class**: This is how my class mcCalculator look like. Its has five functions besides constructor and destructor called Add, Subtract, Devide (spell mistake :)) , Multiply, and DisplayOutVal. As you can see from their names, these functions adds, subtracts,

multiplies, divides correspondingly and stores result in a private variable iOutVal which is called by DisplayOutVal to display the result.

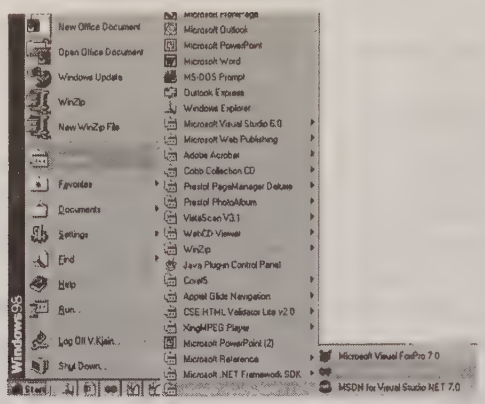
Listing 7.10

```
// four function calculator (cal.cs)
using System;
public class mcCalculator
{
   private int iOutVal; //Default constructor
   public mcCalculator() { iOutVal = 0; }
   public mcCalculator(int iVal1 ) { iOutVal = iVal1 ; } //destructor
   ~mcCalculator() { iOutVal = 0; } //methods
   public void displayiOutVal() { The console.WriteLine("iOutVal =
{0}", iOutVal); }          // this function adds two values and saves
in iOutVal
   public void add(int iVal1, int iVal2 ) { iOutVal = iVal1 + iVal2
;}
   // this function subtracts one from other and saves in iOutVal
   public void subtract(int iVal1, int iVal2 ) { iOutVal = iVal1 -
iVal2 ; }
   // this function multiplyies two values and saves in iOutVal
   public void Multiply(int iVal1, int iVal2 ) { iOutVal = iVal1 *
iVal2 ; }
   // this function devides one value from other and saves in
iOutVal
   public void Divide(int iVal1, int iVal2 ) { iOutVal = iVal1 /
iVal2 ; }
}
```

2. Calling your class from Main: Now call mcCalculator from main.
First create an instance of mcCalculator and then call its member functions. Don't forget to call using System.

```
// Main Program
class mcStart
{
   public static void Main()
   {
         mcCalculator mcCal = new mcCalculator(50);
         mcCal.add(45, 67 );
         mcCal.displayiOutVal();
         mcCal.subtract( 67, 45 );
         mcCal.displayiOutVal();
         mcCal.Multiply( 11, 7 );
         mcCal.displayiOutVal();
         mcCal.Devide( 12, 3 );
         mcCal.displayiOutVal();
   }
}
```

The output on the console shall be:



It does not affect you anyway whether you put the code for main program first or at end.

Listing 7.11

```
using System;
// Creating a simple Calculator class
/// <summary>
/// Summary description for Class1.
/// </summary>
public class CalCulator
{
   // Square function
   public int Square(int num)
   {
        return num*num;
   }
   // Add two integers and returns the sum
   public int Add(int num1, int num2 ) { return num1 + num2; }
   // Add two integers and returns the sum
   public double Add(double num1, double num2 ) { return num1 + num2; }
   // Multiply two integers and retuns the result
   public int Multiply(int num1, int num2 ) { return num1 * num2; }
   // Subtracts small number from big number
   public int Subtract(int num1, int num2 )
   {
        if ( num1 > num2 )
        {
             return num1 - num2;
        }
        return num2 - num1;
```

```
    }
}
// This shows how to use CalCulator class from Main C# program.
public class Class1
{
    static void Main()
    {
          CalCulator sq = new CalCulator();
          The console.WriteLine( sq.Square(6).ToString());
          The console.WriteLine( sq.Add(11.23, 3.67).ToString());
          The console.WriteLine( sq.Multiply(6,9).ToString());
          The console.WriteLine( sq.Subtract(33, 12).ToString());
    }
}
```

7.6 A Simple Calculator Class

This section shows how to create a new class and call its member functions from Main program. CalCulator is a class which has some member functions such as Square, Add, Subtract, Multiply etc.

Listing 7.12

```
using System;
// Creating a simple Calculator class  cal1.cs
/// <summary>
/// Summary description for Class1.
/// </summary>
public class CalCulator
{
    // Square function
    public int Square(int num)
    {
          return num*num;
    } // Add two integers and returns the sum
    public int Add(int num1, int num2 )
    {
          return num1 + num2;
    }
    // Add two integers and returns the sum
    public double Add(double num1, double num2 )
    {
          return num1 + num2;
    }
    // Multiply two integers and retuns the result
    public int Multiply(int num1, int num2 )
{
```

```
return num1 * num2;
} // Subtracts small number from big number
public int Subtract(int num1, int num2 )
{
if ( num1 > num2 )
{
return num1 - num2;
}
return num2 - num1;
}
}
// This shows how to use CalCulator class from Main C# program.
public class Class1
{
static void Main()
{
CalCulator sq = new CalCulator();
The console.WriteLine( sq.Square(8).ToString());
The console.WriteLine( sq.Add(8.3, 9.24).ToString());
The console.WriteLine( sq.Multiply(5,8).ToString());
The console.WriteLine( sq.Subtract(22, 42).ToString());
}
}
```

Build and run the application

Interfaces and Value Data Types

Listing 7.13

```
//sharp6.cs
 class xyz6 {
   static void Main()
   {
       alpha();
   }
}
```

The output of this program will be displayed on the the console as:

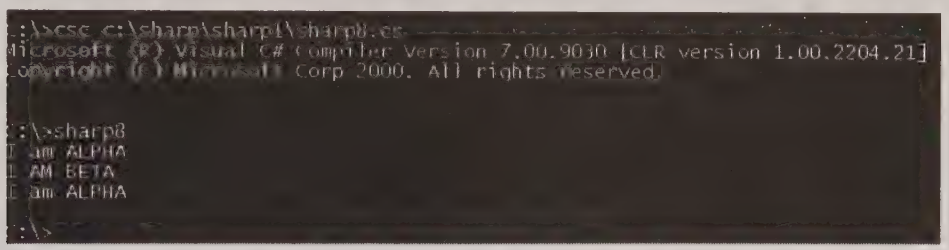
```
C:\>csc c:\sharp\sharp1\sharp6.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp6.cs(6,1): error CS0103: The name 'alpha' does not exist in th
e class or namespace 'xyz6'

C:\>
```

This error message is due to the fact that that we have so far not defined or created the function.There is no function alpha in c# like readymade function System.The console.WriteLine. So let us create this function as shown in next program.

Listing 7.14

```
//sharp8.cs
class xyz7
{
 static void Main()
{
   alpha();   //line 7  juridiction of class xyz7
   static void alpha()   //line 8
  {
   System.The console.WriteLine("C-Sharp");
  }
}
```

The output of this program shall be displayed on the the console as:



In the above program, the function alpha is called upon at line 7 and defined in lines 8 to 12.The function definition is at line 8 and the code included inside the two curly braces of line 9 and 11 is the actual function.The class xyz7 has the jurisdiction over full program and the function. The function alpha() is called from the main function's jurisdiction and defined outside the main function but inside the jurisdiction of class xyz8.

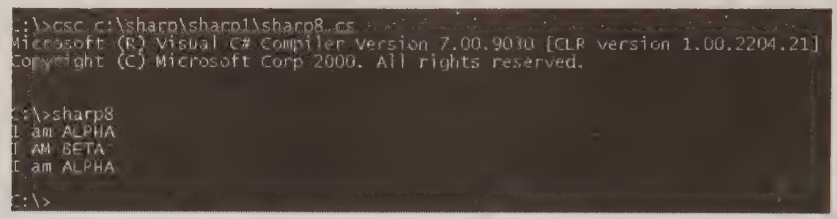
You can call as many functions in a class as required but each one is to be separated from other by ";" as illustrated in the program sharp8.cs below.

Listing 7.15

```
//sharp8.cs
 class xyz8 {
   static void Main()
   {
     alpha();      //line 5
     beta();       //line 6
     alpha();      //line 7
   }
   static void alpha()        //line8
   {
     System.The console.WriteLine("I am ALPHA");
```

```
    }
    static void beta()
    {
      System.The console.WriteLine("I AM BETA");
    }
}
```

The output of this program shall be displayed on the the console as:

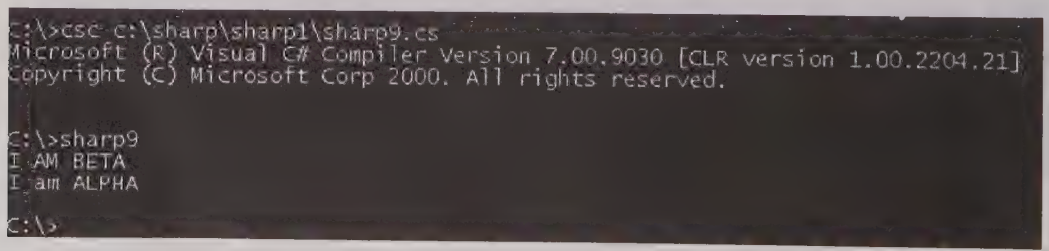


It is noteworthy that output of the program is printed in the sequence in which functions had been called upon. In program sharp9.cs below, the function beta is called from within function alpha.

Listing 7.16

```
//sharp9.cs
 class xyz9 {
   static void Main()
   {
     alpha();
   }
   static void alpha()
   {
     beta();
     System.The console.WriteLine("I am ALPHA");
    }
   static void beta()
   {
     System.The console.WriteLine("I AM BETA");
   }
}
```

The output of this program shall be displayed on the the console as:


C:\>csc c:\sharp\sharp1\sharp9.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>sharp9
I AM BETA
I am ALPHA
C:\>

C++ has the facility (through var args and ...) to allow you to write methods with a variable number of parameters. C# has a similar facility with the params modifier. A parameter marked as such should be the last one in the parameter list and must be a single dimension array, for example:

```
int Sum(params int[] vals) {
   int sum = 0;
   foreach (int i in vals) sum += i;
   return sum;
}

Sum() can be called like this
    The console.WriteLine("Sum is {0}", Sum(1, 2, 3, 4));
```

7.7 Try and Catch

The final aspect that we shall explore here is exceptions. C# supports the try-catch-finally syntax from C++. When execution leaves the code in the try block, the code in the finally block is executed and if the reason for leaving the try block is because an exception was thrown, the catch block is called.

```
try {
   The console.WriteLine("in try");
   throw new Exception("exception thrown");
}
catch(Exception e) {
   The console.WriteLine("in catch: {0}", e.Message);
}
finally {
   The console.WriteLine("in finally");
}
```

This will result in:

```
in try in catch: exception thrown in finally
```

Exceptions that are thrown have to be of type System.Exception, or a class derived from it. As you can see from the code above, if the exception is constructed with a string you can retrieve the string later through the Exception.Message property.

Of course exceptions are used for exceptional conditions, and you really should not use them in situations like this when one of the C# flow statements like **`if`** could be used. One confusing aspect of C# is that there is no equivalent of the C++ or Java throws statement, so you must scrutinize a class's documentation to see if it could throw a specific exception type. If you cannot get this information from the class's documentation then you have no choice but to surround the use of the class with a try block and catch the generic **`System.Exception`** type.

Conversions from one type to another can lose information, for example:

```
uint ui = 0x10000;
ushort us = (ushort)ui;
```

After this(assuming it was compiled with the default settings of csc) **us** will have a value of 0 because a **ushort** has 16 bits and **ui** has 17 bits of information. This may be your intention in doing the cast, but often this is an error. By default, the C# compiler will not check for overflow but this can be changed using the /checked switch. If you use /checked+ then all integer code will be checked for overflow and the previous code will throw a **System.OverflowException** exception. Of course, the switch has an affect on all code in a source file.

To specify that an explicit check is made for overflows on individual code blocks, we can surround that code with a checked block:

```
uint ui = 0x10000; ushort us;
try {
   checked
   {
      us = (ushort)ui;
   }
} catch(Exception e) {
   The console.WriteLine(e.ToString());
}
```

This code will throw a System.OverflowException exception. There is a corresponding unchecked keyword that allows you to turn off overflow checking which is useful when you have compiled a file with /checked+.

Listing 7.17: input3.cs

```
using System;
public class SquareSample {
    public int CalcSquare(int nSideLength)
    {
          return nSideLength*nSideLength;
    }
}
class SquareApp {
    public static void Main()
    {
         SquareSample sq = new SquareSample();
         The console.WriteLine(sq.CalcSquare(25).ToString());
    }
}
```

The output of this program shall be:

```
C:\>input3
625
```

ref Parameters

Aithough you can create many methods using in parameters and return values, you are out of luck as soon as you want to pass a value and have it modified in place(the same memory location) i.e. where the reference parameter comes in handy:

```
void myMethod(ref int nInOut)
```

since you pass a variable to the method (and not its value only),the variable nInOut must be initialized.Otherwise, the compiler will complain. Listing 7.18 shows how to create a method with a ref parameter.

Listing 7.18

```
// input7.cs
class SquareSample
using System;
public class SquareSample {
      public void CalcSquare(ref int nOne4All)
      {
           nOne4All *= nOne4All;
      }
}
class SquareApp {
       public static void Main ()
       {
          SquareSample sq = new SquareSample();
           int nSquaredRef = 20;  // must be initialized
           sq.CalcSquare(ref nSquaredRef);
           The console.WriteLine(nSquaredRef.ToString());
       }
}
```

The output of this program shall be:

```
C:\>csc \input7.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version ---]
Copyright (C) Microsoft Corp 2000. All rights reserved.

fatal error CS0007: Unexpected Common Language Runtime initialization e
Class not registered
```

As you can see, all you have to do is to add the ref modifier to both the definition and the call. Because the variable is passed by reference, you can use it to compute the result and pass back it. However, in a real-world application we should have two variables, one in parameter and one ref parameter.

out Parameters

The third option for passing a parameter is to designate it as an out parameter. As the name implies, an out parameter can be used only to pass a result back from a method. Another difference from the ref parameter is that the caller doesn't need to initialize the variable prior to calling the method. This is shown in Listing 7.19.

Listing 7.19

```
input8.cs
using System;
public class SquareSample {
       public void CalcSquare(int nSideLength, out int nSquared)
       {
            nSquared = nSideLength * nSideLength;
        }
}
class SquareApp {
     public static void Main()
     {
         SquareSample sq = new SquareSample();
          int nSquared;  // need not be initialized
          sq.CalcSquare(15, out nSquared);
          The console.WriteLine(nSquared.ToString());
      }
}
```

Listing 7.20

```
input4.cs
using System;
class Triangle {
       public virtual double ComputeArea(int a, int b, int c)
       {
           // Heronian formula
           double s = (a +b + c) /2.0;
           double dArea = Math.Sqrt(s*(s-a)*(s-b)*(s-c));
           return dArea;
       }
}
class RightAngledTriangle:Triangle {
        public override double ComputeArea (int a, int b, int c)
        {
            double dArea = a*b/2.0;
            return dArea;
        }
}
class TriangleTestApp {
       public static void Main()
       {
           Triangle tri = new Triangle();
            The console.WriteLine(tri.ComputeArea(2, ,5 ,6));
```

```
            RightAngledTriangle rat = new RightAngledTriangle(); //
line 14
            The console.WriteLine(rat.ComputeArea(3, 4, 5));
        }
}
```

The base class Triangle defines the method ComputeArea. It takes three integer parameters, returns a double result, and is publicly accessible. Derived from the class Triangle is RightAngeledTriangle,which overrides the ComputeArea method and implements its own area calculation formula. Both classes are instantiated and tested in the Main()method of the test application class named TriangleTestApp.
Let us understand line 14 in Listing 7.20:

```
class RightAngledTriangle : Triangle
```

The colon (:) in the class statement denotes that RightAngeledTriangle derives from the class Triangle. That is all you have to do to let C# know that you want Triangle as the base class for right angled triangle.
When you take a close look at the ComputeArea method for a right angle triangle, you will observe that the third parameter isn't used for the calculation. However, one can create a "right angleness" check by using the third parameter as shown in Listing 7.20.
The output of this program will be:

```
C:\>input4
4.6837484987987983
6
```

Listing 7.21

```
input5.cs
using System;
class Triangle {
        public virtual double ComputeArea(int a, int b, int c)
        {
            // Heronian formula
            double s = (a +b + c) /2.0;
            double dArea = Math.Sqrt(s*(s-a)*(s-b)*(s-c));
            return dArea;
        }
}
class RightAngledTriangle:Triangle {
         public override double ComputeArea (int a, int b, int c)
         {
             double dArea = a*b/2.0;
             return dArea;
         }
```

```
}
class TriangleTestApp {
        public static void Main()
        {
            Triangle tri = new Triangle();
             The console.WriteLine(tri.ComputeArea(3 ,4 ,5));
             RightAngledTriangle rat = new RightAngledTriangle();
             The console.WriteLine(rat.ComputeArea(2 ,5 ,6));
        }
}
class  RightAngeledTriangle :Triangle {
        public override double ComputeArea(int a, int b, int c)
        {
            const double dEpsilon = 0.0001;
            double dArea = 0;
            if (Math.Abs((a*a + b*b - c*c)) > dEpsilon)
            {
               dArea = base.ComputeArea(a,b,c);
            }
            else
            {
                dArea = a*b/2.0;
            }
            return dArea;
        }
}
```

The check is simply the formula of Pythagoras, which must yield zero for a right-angled triangle. If the result differs from zero (and a delta epsilon), the class calls the ComputeArea implementation of its base class:

```
dArea = base.ComputeArea(a,b,c);
```

The point of the example is that you can easily call the base class implementation of an overridden method explicity using the **base.qualifier**.This is very helpful when you need the functionality implemented in the base class, but don't want to duplicate it in the overridden method.

The output of this program is as follows:

8

STATEMENTS AND CONTROL

It is the judicious use of control statements that generates simple programs for complex problems.Thus, a good programmer should be well conversant with the use of control statements.

This lesson teaches you how to use C# Selection Control Statements. It has been intended to meet the following objectives:

- Learn the "if" statements.
- Learn the "switch" statement.
- Learn how "break" is used in "switch" statements.
- Understand proper use of the "goto" statement.

Iteration (Looping) Statements

- Learn the "while" loop.
- Learn the "do" loop.
- Learn the "for" loop.
- Learn the foreach loop.
- Complete your knowledge of the "break" statement.
- Teach you how to use the "continue" statement.

8.1 Controls and Statements

The performance of a program is almost entirely carried out by control statements.For instance,control statements determine whether or not to execute parts of a program and how many times they are to be repeated. The C++ language has a small but flexible repertoire of such statements. In this chapter, you will read about:

- structured and nonstructured control
- repetitions: while, for, and do-while
- interruptions: break and continue
- conditionals: else and switch

8.2 Statements

C# provides a variety of statements, most of which are similar to those in C and C++ languages.

The following are some varieties of statements:

labeled-statement
declaration-statement
embedded-statement

embedded-statement:

block
empty-statement
expression-statement
selection-statement
iteration-statement
jump-statement
try-statement
checked-statement
unchecked-statement
lock-statement
using-statement

8.3 Selection statements

Selection statements select one of the number of possible statements for execution depending on the value of a controlling expression.

We will consider the following selection-statements:

- if-statement
- switch-statement

8.4 The if statement

In the previous lessons, every program you saw contained a limited amount of sequential steps and a definite point where the program stopped. There were no decisions to be made with the input and the only method was to follow straight through to the end. The information in this lesson with help you branch a program into separate logical sequences based on the decisions you make.

Our first selection statement is the "if" statement. It has three primary forms:

- a single decision
- an either/or decision
- multi-case decision.

The if statement selects a statement for execution based on the value of a boolean expression. As already stated,the if statement gives the programmer a decision block by virtue of which he can incorporate a condition in it in brackets immediately following it , depending upon which the next statement will be executed. In the following case the if statement is for "false", therefore

the code in next line is not executed. After this line there is no executable statement in the program, and therefore the program will terminate with the message "unreachable code detected".

Listing 8.1:

```
//sharp19.cs
  class xyz19 {
   static void Main()
   {
     if(false)
     System.Console.WriteLine("C-Sharp");
   }
}
```

The output of this program shall be displayed on the console as:

```
C:\>csc c:\sharp\sharp1\sharp19.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp19.cs(7,4): warning CS0162: Unreachable code detected

C:\>
```

In the program sharp20.cs, the message in the second printline shall get executed as the the "if" statement evaluates to false. But while compiling you will get the same error message. If we write (true) as the argument for if statement, then the message for both printline will be executed.

Listing 8.2 :

```
//sharp20.cs
  class xyz20 {
   static void Main()
{
   if (false)
   System.Console.WriteLine ("C-Sharp");
   System.Console.WriteLine ("Goodbye");
} }
```

The output of this program shall be displayed on the console as:

```
C:\>sharp30
beta
C-Sharp 786
```

Listing 8.3 :

```
//sharp21.cs
  class xyz21 {
   static void Main()
```

```
{
   if( false )
{
   System.Console.WriteLine("C-Sharp");
   System.Console.WriteLine("Goodbye");
} } }
```

The output of this program will be as follows:

```
C:\>csc c:\sharp\sharp1\sharp21.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp21.cs(8,4): warning CS0162: Unreachable code detected

C:\>sharp21

C:\>
```

The above program clarifies that if you want to affect both the WriteLine statements, then enclose both of them in curly braces. In such a case, the program will skip both the lines and print nothing.

Listing 8.4 :

```
//sharp22.cs
  class xyz22 {
   static void Main()
{
   if(786>790)
   System.Console.WriteLine("C-Sharp");
   System.Console.WriteLine("Goodbye");
} }
```

The output of this program will be displayed on the console as:

```
C:\>csc c:\sharp\sharp1\sharp22.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp22.cs(7,4): warning CS0162: Unreachable code detected

C:\>sharp22
Goodbye

C:\>
```

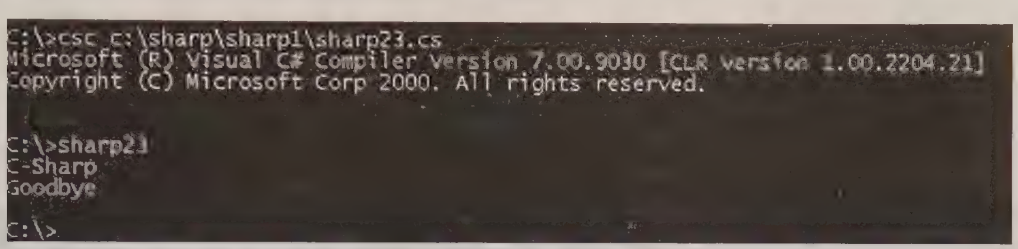
Since the expression 786 > 790 is false, the if statement program has skipped the first WriteLine statement and printed the result as "Goodbye".

Listing 8.5 :

```
//sharp23.cs
  class xyz23 {
   static void Main()
{
```

```
    bool alpha;
    alpha=true;
    if (alpha)
{
    System.Console.WriteLine("C-Sharp");
    System.Console.WriteLine("Goodbye");
} } }
```

The output of this program shall be displayed on the console as:



The above variation of the if statement can be employed to produce results by introduction of boolean variable. The boolean variable alpha is assigned the value of being true in the initialisation statement alpha = true, so the if statement evaluates to be true. This causes both the messages to be printed since both of these statements are enclosed inside the curly braces and are thus affected together.

8.5 The Syntax of the if Statement

```
if-statement:
if   (boolean-expression)   embedded-statement
if   (boolean-expression)  embedded-statement else  embedded-
statement
boolean-expression: expression
```

The if statement is the most extensively used conditional statement in C as well as a number of other languages. The syntax of an if statement is:

```
if (expression)
    statement(s)
```

The expression after the if keyword may be referred to by several names such as the object expression, or the test condition.The statement following the expression is referred to as the object statement or the substatement.
On encountering the if statement, the parenthesized expression is evaluated first. If it is true (evaluates to a non- zero value), then the object statement is executed. If it evaluates to false (zero), then object statement is ignored and control presses to the next statement.

In the following example:

Listing 8.6 :

```
//if3.cs
  using System;
  class Test {
   public static void Main()
{
   int  a ,b ;
   a= 10;
   b= 20;
   if (a < b)
   System.Console.WriteLine("true, a is less than b ");
}
}
```

The output of this program is:

```
C:\>if3
true, a is less than b
```

since the given value of a is less than b, you see the message "True a is less than b". Try rerunning after changing the value of 'a'. The cout statement is executed only if the relational expression (a < b) is true. In this particular example, if the symbol < is replaced by > there will be no output on compiling and running the program.

When the following program

Listing 8.7 :

```
//if4.cs
  using System;
  class Test {
   public static void Main()
 {  int code;
    code = 1;
    if (code == 1)
    {
       System.Console.WriteLine( "The water was too warm\n") ;
       System.Console.WriteLine( "The waters were all fished out\n");
       System.Console.WriteLine( "It was too late in the season\n");
    }
    System.Console.WriteLine( "*** End of fishing excuses ***") ;
 }
}
```

is executed, the output is as follows:

The water was too warm

The waters were all fished out

It was too late in the season

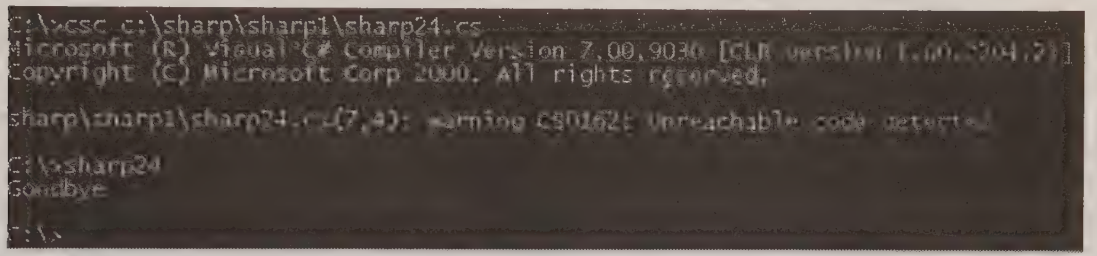
*** End of fishing excuses ***

8.6 If-else statement

Listing 8.8 :

```
//sharp24.cs
  class xyz24 {
   static void Main()
{
   if (false)
   System.Console.WriteLine("C-Sharp");
   else
   System.Console.WriteLine("Goodbye");
} }
```

The output of this program will be:



In the if-else statement first the expression after 'if' is evaluated and if it is true the immediately following statement is evaluated. Otherwise the statement after the "else" is evaluated. In the program above, since the if portion is false, the first WriteLine statement is skipped and the statement after the else portion gets printed.

8.7 Nested ifs

We will develop some of the major concepts about control structures with the aid of the following program. A teacher wants to design a program to read in a list of final scores of a defined number of students and on the basis of these assign the grade obtained by each student.Grading scheme shall be:

Scored marks	Grade
90-100	A
80-<90	B
70-<80	C
60-<70	D
0-<60	E

Furthermore, some students of this class very are designated for auditing the course. It is originally assumed that auditing members of the class do not receive "real" test scores and that the program need only handle one set of input values at a time. These two limiting assumptions will be relaxed for the time being and additional simple numeric manipulation will be included later in this chapter. Ordinarily the object expression involves an equality or relational test. For example, in this case, decision steps could be written as follows:

Program segment 1

```
if (score >= 0)
   grade = 'E';
if (score > 60)
   grade = 'D';
if (score > 70)
   grade = 'C';
if (score > 80)
   grade = 'B';
if (score > 90)
   grade = 'A';
if (score == 999)
   grade = 'T';
```

The grade T is awarded to the auditing students, and the program section assumes that the auditing students were assigned a score of 999 to differentiate them from the other students. The direct assignment of T to the variable score would cause ambiguity because the ASCII value of T is decimal 84.Besides,a value of 84 can also represent a grade of B. Remember, in C the integer and character classes are very similar. Note that not only is score involved in six relating tests but that during execution of this one section of code, the variable grade could be assigned a character value and then reassigned another character value.For instance, consider a value for score 72. Since this value is greater than zero, grade is initially assigned a character value of E during the first if decision construction. Flow then passes into the second if statement where score is found to be greater than 60.Grade is then reassigned a value of character D. Control flow then passes to the third if construction where grade is assigned a value of C. After this statement, score no longer causes the remaining condition expression to evaluate to true so no further reassignment occur. We shall examine this problem in detail section 5. of this unit.

Multiple if statements should not be used in this manner for four reasons:

- Execution time is wasted executing unnecessary object statement
- Execution time is wasted evaluating redundant multiple relational and/or equality expressions.
- This construction is only applicable to steadily increasing or decreasing categories and only then will the object statement be reversible (i.e. a printf() would be irreversible).
- Multiple if construction (used on non-mutually exclusive categories - see below) are confusing and highly error prone.

This type of construction should only be used for mutually exclusive categories. Category B is mutually exclusive of category C if the inclusion of an item in C precludes it's inclusion in B (and vice-versa). As it stands, the example does not contain mutually exclusive categories. A score can "trip" (or cause a yes value) in more than one if statement.

Example

The if statement requires that the condition inside the if statement evaluate to ban expression of type bool.

Listing 8.9 :

```
//if2.cs
  using System;
  class Test {
   public static void Main()
{
   int   value = 0;

   if (  value  == 0)
   System.Console.WriteLine("true");
}
}
```

The output of this program is :

```
C:\>csc \if1.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

if1.cs(9,8): error CS0029: Cannot implicitly convert type 'int' to 'bool'
```

The following program will give the error message:

Listing 8.10 :

```
//if2.cs
  using System;
  class Test {
   public static void Main()
{
   int   value;
   value = 0;
 if (  value )  //invalid
   System.Console.WriteLine("true");
  if (  value  == 0)    //must use this
   System.Console.WriteLine("true");
} }
```

```
C:\>csc \if1.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

if1.cs(9,8): error CS0029: Cannot implicitly convert type 'int' to 'bool'
```

8.8 if-then-else7

Listing 8.11 :

```
//else1.cs
  using System;
  class Test {
   public static void Main()
{  int i ;
       i = 3 ;
        Console.WriteLine(  "i ={0} " , i);
       if ( i > 4 )
 Console.WriteLine(  "i is greater than 4" ) ;        // line 1
       else
 Console.WriteLine( "i is not greater than 4")  ;

} }
```

```
C:\>else1
i =3
i is not greater than 4
```

Listing 8.12 :

```
//else2.cs
  using System;
  class Test {
   public static void Main()
{
   int code;
   code = 1;
   if ( code == 1 )
    Console.WriteLine( "The water was too warm\n") ;
   else
    Console.WriteLine( "The waters were all fished out\n") ;
         Console.WriteLine( "*** End of fishing excuses ***") ; } }
```

The output will be :

```
C:\>else2
The water was too warm

*** End of fishing excuses ***
```

This is a good example to bring out that incase the else statement is not executed computer takes up the next line for execution, that is, the last line in this case.

Listing 8.13 :

```
//else3.cs
  using System;
  class Test {
   public static void Main()
{
   int code;
   code = 1;
 {  if ( code != 1 )
    Console.WriteLine( "The water was too warm\n") ;

else
    Console.WriteLine( "The waters were all fished out\n") ;
}
        Console.WriteLine( "*** End of fishing excuses ***") ;
}
}
```

The output of this program will be :

```
C:\>else3
The waters were all fished out

*** End of fishing excuses ***
```

An else part is associated with the nearest preceding the **if** statement that does not already have an else part. That is, an if statement of the form:

```
if (x) if (y) F(); else G();
```

is equivalent to

```
if (x) {
     if (y) {
          F();
     }
     else {
          G();
     }
}
```

An if-else statement is executed as follows:

- The boolean-expression is evaluated.
- If the boolean expression yields true, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the if statement.
- If the boolean expression yields false and if an else part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the if statement.
- If the boolean expression yields false and if an else part is not present, control is transferred to the end point of the if statement.

The first embedded statement of an if statement is reachable if the if statement is reachable and the boolean expression does not have the constant value false.

The second embedded statement of an if statement, if present, is reachable if the if statement is reachable and the boolean expression does not have the constant value true.

The end point of an if statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an if statement with no else part is reachable if the if statement is reachable and the boolean expression does not have the constant value true.

Listing 8.14 : Forms of the IF statement: Sel11.cs

```
using System;
class IfSelect {
    public static void Main() {
        string myInput;
        int myInt;
        Console.Write("Please enter a number: ");
        myInput = Console.ReadLine();
        myInt = Int32.Parse(myInput);
        // Single Decision and Action with brackets
        if (myInt > 0) {
            Console.WriteLine("Your number {0} is greater than
             zero.", myInt);
        }
        // Single Decision and Action without brackets
        if (myInt < 0)
            Console.WriteLine("Your number {0} is less than
             zero.", myInt);
        // Either/Or Decision
        if (myInt != 0) {
            Console.WriteLine("Your number {0} is not equal to
              zero.", myInt);
        }
        else {
            Console.WriteLine("Your number {0} is equal to
              zero.", myInt);
```

```
        }
        // Multiple Case Decision
        if (myInt < 0 || myInt == 0) {
            Console.WriteLine("Your number {0} is less than or
             equal to zero.", myInt);
        }
        else if (myInt > 0 && myInt <= 10) {
            Console.WriteLine("Your number {0} is between 1 and
             10.", myInt);
        }
        else if (myInt > 10 && myInt <= 20) {
            Console.WriteLine("Your number {0} is between 11 and
              20.", myInt);
        }
        else if (myInt > 20 && myInt <= 30) {
            Console.WriteLine("Your number {0} is between 21 and
             30.", myInt);
        }
        else {
            Console.WriteLine("Your number {0} is greater than
              30.", myInt);
        }
    }
}
```

The output of this program will appear similar to what is displayed on console below:

```
C:\>sel11
Please enter a number: 34
Your number 34 is greater than zero.
Your number 34 is not equal to zero.
Your number 34 is greater than 30.

C:\>sel11
Please enter a number: 0
Your number 0 is equal to zero.
Your number 0 is less than or equal to zero.

C:\>sel11
Please enter a number: -34
Your number -34 is less than zero.
Your number -34 is not equal to zero.
Your number -34 is less than or equal to zero.
```

The different situations had been tested—one with value as 34, another with value as 0 and third one as negative.

The statements in Listing 8.14 uses the same input variable, "myInt" as a part of their evaluation. This is another way of obtaining interactive input from the user. We first print the line "Please enter a number: " to the console. The "Console.ReadLine()" statement causes the program to wait for input from the user.The user types a number and then presses the enter or return key. This number is returned in the form of a string into the "myInput" variable, which is a string type. Since we must evaluate the user's input in the form of an integer, "myInput" must be converted. This is done with the command "Int32.Parse(myInput)". (Int32 and similar types will be covered in another lesson on advanced types) The result is placed into the "myInt" variable, which is an integer type.

Now that we have a variable in the type we wanted, we will evaluate it with "if" statements. The first statement is of the form if (boolean expression) { statements }. You must begin with the keyword "if". Next is the boolean expression between parenthesis. This boolean expression must evaluate to a true or false value. Here, we are checking the user's input to see if it is greater than (>) 0. If this expression evaluates to true, we execute the statements within the curly braces. (We refer to the structure with curly braces as a "block") There could be one or more statements within in this block. If the boolean expression evaluates to false, we ignore the statements inside the block and continue the program execution with the next statement after the block.

The second "if" statement is much like the first, except it does not have a block. Therefore, if it's boolean expression evaluates to true, the first statement after the boolean expression will be executed. When the boolean expression evaluates the false, the first statement after the boolean expression will be skipped and the next program statement will be executed. This form of "if" statement is adequate when you only have a single statement to execute. If you want to execute two or more statements when the boolean expression evaluates to true, you must enclose them in a block. Generally speaking, it is better to always put your if statements within a block, regardless of whether or not you have just one statement to execute. This will help avoid mistakes where you later decide to add a statement and may forget to add the curly braces.Very often we have to make an either/or kind of decision. The third "if" statement in Listing 8.14 is for this purpose. When the boolean expression evaluates to true, the statement(s) immediately following the "if" statement are executed. When the boolean expression evaluates to false, the statements following the "else" keyword are executed.

When you have multiple expressions to evaluate, you can use the if/else if/else form of the "if" statement. The fourth "if" statement of Listing 8.14 is of this kind. You begin with the "if" keyword, again executing the following block if the boolean expression evaluates to true. However, this time you can evaluate multiple subsequent conditions with the "else if" keyword combination. The "else if" statement also takes a boolean expression, just like the "if" statement. The rules are the same, when the boolean expression for the "else if" statement evaluates to true, the block immediately following the boolean expression is executed. This can go on to evaluate all cases. Note that the entire "if/else if" sequence must end with a final "else" part. When none of the other "if" or "else if" boolean expressions evaluate to true, the block following the "else" keyword will be executed. Only one section of an if/else if/else statement will be executed.

The last statement is different from the others with regard to the boolean expressions. The boolean expression, (myInt < 0 || myInt == 0), contains the conditional OR (||) operator. In both the regular OR (|) operator and the conditional OR (||) operator, the boolean expression

will evaluate to true if either of the two sub-expressions on either side of the operator evaluate to true. The primary difference between the two OR forms are that the regular OR operator will evaluate both sub-expressions every time. However, the conditional OR will evaluate the second sub-expression only if the first sub-expression evaluates to false.

The boolean expression, (myInt > 0 && myInt <= 10), contains the conditional AND operator. Both the regular AND (&) operator and the conditional AND (&&) operator will return true when both of the sub-expressions on either side of the operator evaluate to true. The difference between the two is that the regular AND operator will evaluate both expressions every time. However, the conditional AND operator will evaluate the second sub-expression only when the first sub-expression evaluates to true.The conditional operators (&& and | |) are commonly called short-circuit operators because they do not always evaluate the entire expression. Thus, their inclusion in the code eliminates unnecessary logic.

8.9 The Switch Statement

Similar to the if/else if/else form of the "if" statement is the "switch" statement.

This statement executes the statements that are associated with the value of the controlling expression.

Listing 8.15 :

```
//sel12.cs
  using System;
  class SwitchSelect {
    public static void Main() {
        string myInput;
        int myInt;
        begin:
Console.Write("Enter an integer number between 1 and 3: ");
        myInput = Console.ReadLine();
        myInt = Int32.Parse(myInput);
        // switch with integer type
        switch (myInt) {
            case 1:
Console.WriteLine("Number as entered by you is{0}.", myInt);
                break;
            case 2:
Console.WriteLine("Number as entered by you is {0}.", myInt);
                break;
            case 3:
Console.WriteLine("Number as entered by you is {0}.", myInt);
                break;
            default:
Console.WriteLine("Number {0} is not between 1 and 3.", myInt);
        }
        decide:
```

```
        Console.Write("Type \"yes\" to continue or \"no\" to quit: ");
             myInput = Console.ReadLine();
             // switch with string type
             switch (myInput) {
                 case "yes":
                     goto begin;
                 case "no":
                     Console.WriteLine("Thank you for trying");
                     break;
                default:
        Console.WriteLine("Your input {0} is incorrect.", myInput);
                     goto decide;
             }
         }
    }
```

The interactive output of program will be as follows:

```
C:\>se112
Enter an integer number between 1 and 3: 2
Number as entered by you is 2.
Type "yes" to continue or "no" to quit: yes
Enter an integer number between 1 and 3: 3
Number as entered by you is 3.
Type "yes" to continue or "no" to quit: yes
Enter an integer number between 1 and 3: 9
Number 9 is not between 1 and 3.
Type "yes" to continue or "no" to quit: quit
Your input quit is incorrect.
Type "yes" to continue or "no" to quit: no
Thank you for trying
```

The statement list of a given switch section is reachable if the switch statement is reachable and at least one of the following is true:

- The switch expression is a non-constant value.
- The switch expression is a constant value that matches a case label in the switch section.
- The switch expression is a constant value that doesn't match any case label, and the switch section contains the default label.
- A switch label of the switch section is referenced by a reachable goto case or goto default statement.

The end point of a switch statement is reachable if at least one of the following is true:

- The switch statement contains a reachable break statement to exit from the switch statement.
- The switch statement is reachable, the switch expression is a non-constant value, and no default label is present.

- The switch statement is reachable, the switch expression is a constant value that doesn't match any case label, and no default label is present.

8.9.1 Syntax of switch Statement

```
switch-statement:
switch (expression)  switch-block
switch-block:
{ switch-sectionsopt } switch-sections: switch-section switch-sections
switch-section switch-section: switch-labels  statement-list switch-
labels: switch-label switch-labels  switch-label switch-label: case
constant-expression  :
default :
```

A switch-statement consists of the keyword switch, followed by a parenthesized expression (called the switch expression), followed by a switch-block. The switch-block consists of zero or more switch-sections, enclosed in braces. Each switch-section consists of one or more switch-labels followed by a statement-list.

The governing aspect of a switch statement is acheived by the switch expression. If the type of the switch expression is sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or an enum-type, then that is the governing type of the switch statement. Otherwise, exactly one user-defined implicit conversion (õ6.4) must exit from the type of the switch expression to one of the following possible governing types: sbyte, byte, short, ushort, int, uint, long, ulong, char, string. If no such implicit conversion exists, or if more that one such implicit conversion exists, a compile-time error occurs.

The constant expression of each case label must denote a value of a type that is implicitly convertible (õ6.1) to the governing type of the switch statement. A compile-time error occurs if two or more case labels in the same switch statement specify the same constant value. There can be at most one default label in a switch statement.

A switch statement functions as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a case label is equal to the value of the switch expression, control is transferred to the statement list that follows the matched case label.
- If no constant matches the value of the switch expression and if a default label is present, control is transferred to the statement list following the default label.
- If no constant matches the value of the switch expression and if no default label is present, control is transferred to the end point of the switch statement.

If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the "no fall through" rule.

```
switch (i) { case 0:
     CaseZero();
     break;
case 1:
```

```
        CaseOne();
        break;
    default:
        CaseOthers();
        break;
    }
```

The above code block is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section in C# cannot "fall through" to the next switch section.The statement block that follows is in error.

```
    switch (i) { case 0:
        CaseZero();
    case 1:
        CaseZeroOrOne();
    default:
        CaseAny();
    }
```

When execution of a switch section is to be followed by execution of another switch section, an explicit goto case or goto default statement must be used:

```
    switch (i) { case 0:
        CaseZero();
        goto case 1;
    case 1:
        CaseZeroOrOne();
        goto default;
    default:
        CaseAny();
        break;
    }
```

Multiple labels are permitted in a switch-section. For example,

```
    switch (i) { case 0:
        CaseZero();
        break;
    case 1:
        CaseOne();
        break;
    case 2: default:
        CaseTwo();
        break;
    }
```

is legal. The example does not violate the "no fall through" rule because the labels case 2: and default: are part of the same switch-section.

The "no fall through" rule ensures that the common class of bugs that occur in C and C++ when break statements are accidentally omitted is eliminated. Also, this rule permits the switch

sections of a switch statement to be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the switch statement above can be reversed without affecting the behavior of the statement:

```
switch (i) { default:
     CaseAny();
     break;
case 1:
     CaseZeroOrOne();
     goto default;
case 0:
     CaseZero();
     goto case 1;
}
```

The statement list of a switch section typically ends in a break, goto case, or goto default statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a while statement controlled by the boolean expression true is known to never reach its end point. Likewise, a throw or return statement always transfer control elsewhere and never reaches its end point. Thus, the following example is valid:

```
switch (i) { case 0:
     while (true) F();
case 1:
     throw new ArgumentException();
case 2:
     return;
}
```

The governing type of a switch statement may be of the type string.
For example:

```
void DoCommand(string command)
{
     switch (command.ToLower())
    {
      case "run":
           DoRun();
           break;
      case "save":
           DoSave();
           break;
      case "quit":
           DoQuit();
           break;
      default:
           InvalidCommand(command);
           break;
      }
}
```

Like the string equality operators, the switch statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a case label constant. As illustrated by the example above, a switch statement can be made case insensitive by converting the switch expression string to lower case and writing all case label constants in lower case.

When the governing type of a switch statement is string, the value null is permitted as a case label constant. A switch-block may contain declaration statements. The scope of a local variable or constant declared in a switch block extends from the declaration to the end of the switch block. Within a switch block, the meaning of a name used in an expression context must always be the same.

8.10 Loop control

8.10.1 Types of Loops

There are two essentially different ways of knowing when a loop has been repeated enough times:

1. When you know exactly how many loops you want to make. An example of this would be if you wished this program to print the word Hello exactly 7 times. We will call a loop of this kind a COUNT -LOOP.
2. When some condition has been fulfilled. This condition might be that a set of numbers should be written by the computer on the screen, one after another until the number 99 is reached. Then the looping should stop. We will call this a CONDITION- LOOP.

8.10.2 Counters.

To control a loop, a special variable can be created. It keeps track of how many times the loop has been executed. Such a variable is called a Counter. The counter increases,by a fixed amount (usually 1) every time the loop is executed. When the programmer knows in advance how many times the loop should be repeated the counter can be tested by an IF/THEN statement after each loop execution to see if the proper number has been reached. To set up a counter for loop control, you should perform the following steps: 1.Initialize the counter (before entering the loop) by setting it to a beginning value. 2. Increment the counter each time the loop is executed. 3. Text the counter each time the loop is executed to see if the loop has been performed the desired number of times.

8.10.3 Loop Control Variable

The loop is an extremely powerful and vital programming tool. Although looping methods are specific of programming languages, these methods share some basic components. A loop control variable, for example, is a variable whose value is used to determine the number of times a loop is repeated. The counter variable is an example of a loop control variable. This is mostly designated by either i ,k or l,m or n or words like count. All loops contain some action that that is repeated.The statements that perform such an action make up the loop body. Execution of the basic loop structure consists of the following five steps:

1. The loop variable is initialized to a particular value before loop execution begins.
2. The program tests the loop control variable to determine whether it should execute the loop body or exit the loop.
3. The loop body, which can consist of any number of statements, is executed.
4. At some point during loop execution, the value of the loop control variable must be modified to allow exit from the loop.
5. The loop is exited when the test in Step 2 determines that the right number of loop repetitions has been made. Execution continues with the next statement following the loop

8.11 Iteration Statements

Iteration statements repeatedly execute an embedded statement. iteration-statement:

- while-statement
- do-statement
- for-statement
- foreach-statement

8.12 The While Statement

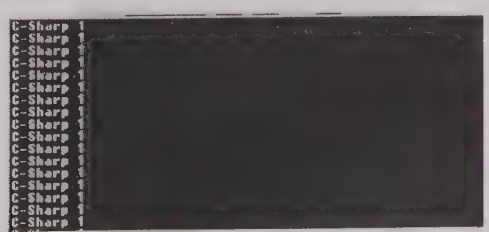
The while statement conditionally executes an embedded statement zero or more times.

Since both the WriteLine statements are enclosed inside the curly braces, both are affected and results are displayed twice.

Listing 8.16 :

```
//sharp28.cs
class xyz28 {
   static void Main()
   {
     int alpha;
     alpha=1;
     while(alpha<=5)
     {
        System.Console.WriteLine("C-Sharp {0}",alpha);
     }
   }
}
```

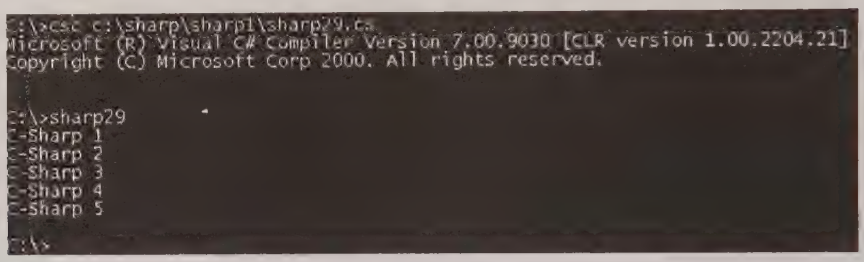
The output of this program shall be displayed on console as:



Listing 8.17 :

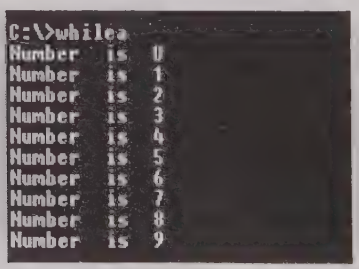
```
//sharp29.cs
class xyz29 {
   static void Main()
{
   int alpha;
   alpha=1;
   while(alpha<=5)
{
   System.Console.WriteLine("C-Sharp {0}",alpha);
   alpha++;
} } }
```

The output of this program shall be displayed on console as:



Listing 8.18 :

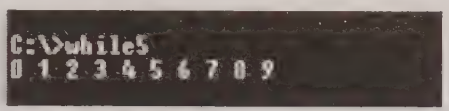
```
//whilea
using  System;
class  Test {
   public  static  void  Main()
    {
      int  n  =  0;
      while  (n  <  10)
       {
           Console.WriteLine("Number  is  {0}",  n);
            n++;
        }
     }
}
```



Listing 8.19 :

```
//while5.cs
using System;
class WhileLoop {
    public static void Main() {
        int myInt = 0;
        while (myInt < 10) {
            Console.Write("{0} ", myInt);
            myInt++;
        }
        Console.WriteLine();
    }
}
```

The output of this program shall be displayed on console as:



Listing 8.19 shows a simple while loop. It begins with the keyword "while", followed by a boolean expression. All control statements use boolean expressions. This means that the expression must evaluate to either a true or false value. In this case we are checking the myInt variable to see if it is less than (<) 10. Since myInt was initialized to 0, the boolean expression will return true the first time it is evaluated. When the boolean expression evaluates to true, the block immediately following the boolean expression will be executed.

Within the while block we print the number and a space to the console. Then we increment (++) myInt to the next integer. Once the statements in the while block have been executed, the boolean expression is evaluated again. This sequence will continue until the boolean expression evaluates to false. Once the boolean expression is evaluated as false, program control will jump to the first statement following the while block. In this case, we will write the numbers 0 through 9 to the console, exit the while block, and print a new line to the console.

Listing 8.20 : Summation of Natural Numbers

```
//sum1.cs
  using System;
  class Summation {
   public static void Main()
 { int sum ,number ;
   sum = 0;                                    // line 1
   number =1;                                  // line 2
   while (number <= 10)                        // line 3
   {                                           // line 4
     sum += number ;                           // line 5
     number++ ;                                // line 6
   }                                           // line 7
   System.Console.WriteLine( "sum = {0}" , sum );      // line 8
 }
}
```

The output of this program shall be displayed on console as: 55

The first time line 3 executes, number equals 1. Since 1 is <= to 10, the body of the while execute, adding number to sum and then incrementing number, Control is then transferred back to line 3. Now number has the value of 2 which is also <= to 10, therefore, the body executes again. When the body of the while has executed 10 times, the value of number is 11 when control is transferred back to line 3. Now number 11 is not <= to 10 and control is passed to line 8 and the while statement is complete.

The break statement can be used to exit the while loop and a continue statement may be used to skip to the immediate closing brace of the while block for this iteration and continue to next operation

Listing 8.21 :

```
using  System;
class  Alpha {
  public  static  void  Main()
  {
    int  n  =  0;
       while  (n  <  10)
       {
          if  (n  ==  3)
          {
             n++;
             continue;
          }
          if  (n  ==  8)
              break;
              Console.WriteLine("Number  is  {0}",  n);
              n++;
       }
  }
}
```

```
C:\>w1
Number  is  0
Number  is  1
Number  is  2
Number  is  4
Number  is  5
Number  is  6
Number  is  7
```

8.12.1 Syntax of While Statement

While (boolean-expression) embedded-statement

A while statement is executed as follows:

1. The boolean-expression (õ8.16) is evaluated.

2. If the boolean expression yields true, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), control is transferred to the beginning of the while statement.
3. If the boolean expression yields false, control is transferred to the end point of the while statement.

Within the embedded statement of a while statement, a break statement (õ8.9.1) may be used to transfer control to the end point of the while statement (thus ending iteration of the embedded statement), and a continue statement (õ8.9.2) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the while statement). The embedded statement of a while statement is reachable if the while statement is reachable and the boolean expression does not have the constant value false. The end point of a while statement is reachable if at least one of the following is true: * The while statement contains a reachable break statement that exits the while statement. * The while statement is reachable and the boolean expression does not have the constant value true.

8.13 The Do Statement

The do loop is very similar to the while loop. The do statement conditionally executes an embedded statement one or more times. The do loop functions just like a while loop, only that the condition is evaluated at the end of the loop and not at the beginning of the loop:

Listing 8.22 :

```
using System; class Gamma {
   public static void Main()
   {
     int n = 0;
     do
     {
        Console.WriteLine("Number is {0}", n);
        n++;
     } while (n < 10);
   }
}
```

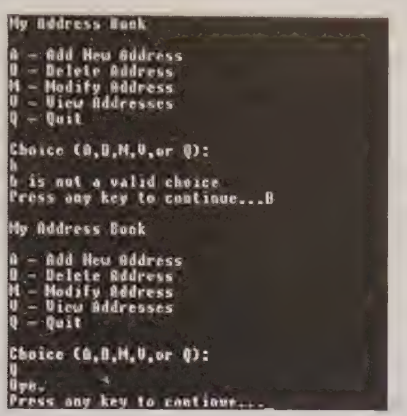
Here also we may use break and continue statements.

Listing 8.23 :

```
//do2.cs
  using System;
  class DoLoop {
    public static void Main() {
        string myChoice;
        do {
            // Print A Menu
            Console.WriteLine("My Address Book\n");
            Console.WriteLine("A - Add New Address");
```

```
        Console.WriteLine("A - Add New Address");
        Console.WriteLine("D - Delete Address");
        Console.WriteLine("M - Modify Address");
        Console.WriteLine("V - View Addresses");
        Console.WriteLine("Q - Quit\n");
        Console.WriteLine("Choice (A,D,M,V,or Q): ");
        // Retrieve the user's choice
        myChoice = Console.ReadLine();
        // Make a decision based on the user's choice
        switch(myChoice) {
            case "A":
            case "a":
         Console.WriteLine("You wish to add an address.");
                break;
            case "D":
            case "d":
        Console.WriteLine("You wish to delete an address.");
                break;
            case "M":
            case "m":
         Console.WriteLine("You wish to modify an address.");
                break;
            case "V":
            case "v":
        Console.WriteLine("You wish to view the address list.");
                break;
            case "Q":
            case "q":
                Console.WriteLine("Bye.");
                break;
            default:
    Console.WriteLine("{0} is not a valid choice",myChoice);
        }
        // Pause to allow the user to see the results
        Console.Write("Press any key to continue...");
        Console.ReadLine();
        Console.WriteLine();
    } while (myChoice != "Q" && myChoice != "q");
     // Keep going until the user wants to quit
```

The following may be the interactive output of above program



Listing 8.23 shows a "do" loop in action. The syntax of the "do" loop is "do { <statements> } while (<boolean expression>);". The statements can be any valid C# programming statements. The boolean expression is the same as for the others we have encountered so far. It returns either true or false.

One reason you may want to use a "do" loop instead of a "while" loop is to present a message or menu such as the one in Listing 8.23 and then retrieve input from a user. Since the boolean expression is evaluated at the end of the loop, the "do" loop guarantees that the statement's inside the loop will execute at least once. However, since a "while" loop evaluates it's boolean expression at the beginning, it is not necessary that the statements inside the loop will be executed, unless programmed explicitly for this.

Let us take an overall view of the Listing 8.14. In the Main() method, we declare the variable "myChoice" of type string. Then we print a series of statement to the console. This is a menu of choices for the user. We must get input from the user, which is in the form of a Console.ReadLine() method which returns the user's value into the myChoice variable. We must take the user's input and process it. This is conveniently done with a "switch" statement. Note that we have placed matching upper and lower case letters together to obtain the same functionality. This is the onlyavailable choice to effect automatic fall through between cases. If you place any statement between the two cases, you would not be able to fall through. Another point is that we used the "default:" case—a good general practice.

8.13.1 The Syntax of do-statement:

do embedded-statement while (boolean-expression);

A do statement is executed as follows:

1. Control is transferred to the embedded statement.
2. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), the boolean-expression is evaluated. If the boolean expression yields true, control is transferred to the beginning of the do statement. Otherwise, control is transferred to the end point of the do statement.

Within the embedded statement of a do statement, a break statement (õ8.9.1) may be used to transfer control to the end point of the do statement (thus ending iteration of the embedded statement), and a continue statement (õ8.9.2) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the do statement).

The embedded statement of a do statement is reachable if the do statement is reachable. The end point of a do statement is reachable if at least one of the following is true:

1. The do statement contains a reachable break statement that exits the do statement.
2. The end point of the embedded statement is reachable and the boolean expression does not have the constant value true.

8.14 The For Statement

The for statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

The for statement includes three arguments separated by comma within its pair of brackets. The first one (alpha) initialises the variable to the value from which loop is to be started and second one (alpha <= 5) tests a condition upto which this statement will cause the program to repeat the loop. The third argument (alpha++) puts the desired increment to the variable for going to next statement. In the program given below the for statement tells the computer to print the message for value of variable alpha from 1 to 5 times.

Listing 8.22

```
//sharp25.cs
  class xyz25 {
   static void Main()
{
   int alpha;
   for (alpha = 1 ; alpha < = 5 ; alpha++)
   System.Console.WriteLine("C-Sharp {0}",alpha);
}
}
```

The output of this program shall be displayed on console as:

```
C:\>csc c:\sharp\sharp1\sharp25.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.


C:\>sharp25
C-Sharp 1
C-Sharp 2
C-Sharp 3
C-Sharp 4
C-Sharp 5

C:\>
```

Listing 8.23

```
//sharp26.cs
  class xyz26 {
   static void Main()
{
   int alpha;
   for (alpha=1;alpha<=5;alpha++)
   System.Console.WriteLine("C-Sharp {0}",alpha);
   System.Console.WriteLine("C-Sharp {0}..",alpha);
}
}
```

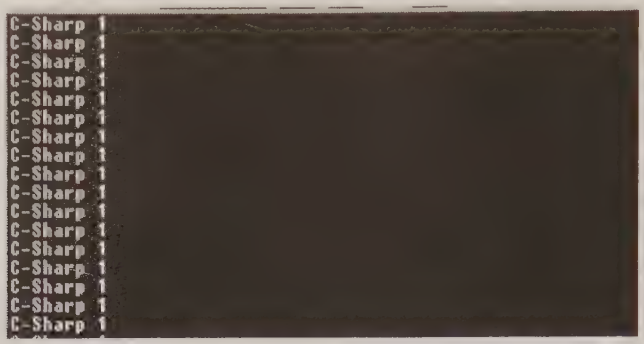
The output of this program shall be displayed on console as:

In this program, the message is printed five times by virtue of the for statement and first WriteLine statement and when the for loop is exhausted the sixth line is printed due to next WriteLine statement.

Listing 8.24

```
//sharp28.cs
  class xyz27 {
   static void Main()
{
   int alpha;
   for (alpha = 1 ; alpha <=5 ; alpha++)
{
   System.Console.WriteLine("C-Sharp {0}",alpha);
   System.Console.WriteLine("C-Sharp {0}..",alpha);
}
}
}
```

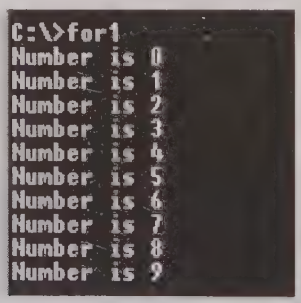
The output of this program shall be displayed on console as:



Listing 8.25

```
//for1.cs
    using System;
    class Test {
     public static void Main()
{

   for (int n = 0; n < 10; n++)


   Console.WriteLine("Number is {0}", n);
}
}
```

The output of this program shall be displayed on console as:

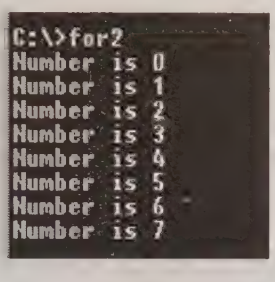


Listing 8.26

```
//for2.cs
using System;
class Test {
   public static void Main()
```

```
    {
     for (int n = 0; n < 10; n++)
      { if (n == 8)
         break;
     Console.WriteLine("Number is {0}", n);
    }
  }
}
```

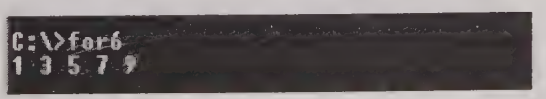
The output of this program shall be displayed on console as:



Listing 8.27

```
using System;
class ForLoop {
    public static void Main() {
        for (int i=0; i < 20; i++) {
            if (i == 10)
                break;
            if (i % 2 == 0)
                continue;
            Console.Write("{0} ", i);
        }
        Console.WriteLine();
    }
}
```

The output of this program shall be displayed on console as:



Listing 8.27 shows the "for" loop. For loops are best used when you know exactly how many times you want to perform the statements within the loop. This listing produces the same results as the "while" loop in Listing 4-1. The contents within the "for" loop parenthesis holds three sections separated by semicolons "(<initializer list>; <boolean expression>, <post-loop action list>)".

The initializer list is a comma separated list of expressions. These expressions are evaluated only once during the lifetime of the "for" loop. This is in the beginning, before loop execution. As shown in Listing 4-3, this section is commonly used to initialize an integer to be used as a counter.

Once the initializer list has been evaluated, the "for" loop transfers control to its second section, the boolean expression. There is only one boolean expression, but it can be as complicated as you may please as long as the result is true or false. The boolean expression is commonly used to verify the status of a counter variable.

When the boolean expression evaluates to true, the statements within the curly braces of the "for" loop are executed. Normally, these statements execute from the opening curly brace to the closing curly brace without interruption. However, in Listing 4-3, we've made a couple exception. We have a couple "if" statements that disrupt the flow of control within the "for" block.

The first "if" statement checks to see if "i" is equal to 10. Now let us consider another use of the "break" statement. It's behavior is similar to the selection statements. It simply breaks out of the loop at that point and transfers control to the first statement following the end of the "for" block.

The second "if" statement uses the remainder operator to see if "i" is a multiple of 2. This will evaluate to true when "i" is divided by 2 with a remainder equal to zero, (0). When true, the "continue" statement is executed. Control will skip over the remaining statements in the loop and transfer back to the post- loop action list. By arranging the statements within a block properly, you can conditionally execute them based upon the condition you need.

When the program control encounters either a continue statement or end of block, right curly brace, it transfers to the third section within the "for" loop parentheses, the post-loop action list. This is a comma separated list of actions you would like to see occur after the statements in the "for" block have been executed. Listing 4-3 is a typical action, incrementing the counter. Once this is complete, control transfers to the boolean expression for evaluation. The loop will continue as long as the boolean expression is true. When the boolean expression becomes false, control is transferred to the first statement following the "for" block.

8.14.1 Syntax of for Statement

for-statement:

```
for (for-initializeropt; for-conditionopt; for-iteratoropt) embedded-
statement for-initializer: local-variable-declaration statement-
expression-list for-condition: boolean-expression for-iterator:
statement-expression-list statement-expression-list: statement-expression
statement-expression-list, statement-expression
```

The for-initializer, if present, consists of either a local-variable-declaration or a list of statement-expressions separated by commas. The scope of a local variable declared by a for-initializer starts at the variable-declarator for the variable and extends to the end of the embedded statement.The scope covers the for-condition and the for-iterator. The for-condition, if present, must be a boolean-expression. The for-iterator, if present, consists of a list of statement-expressions (õ8.6) separated by commas.

A for statement is executed as follows:

1. If a for-initializer is present, the variable initializers or statement expressions are xecuted in the order they are written. This step is performed only once.
2. If a for-condition is present, it is evaluated.
3. If the for-condition is not present or if the evaluation yields true, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), the expressions of the for-iterator, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the for-condition in the step above.
4 If the for-condition is present and the evaluation yields false, control is transferred to the end point of the for statement.

8.15 The foreach Statement

Foreach is a statement which acts on a collection type. The foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection. A collection type is something that implements the IEnumerable interface. This interface has a single method called GetEnumerator() that returns the IEnumerator interface on an object that will be called to enumerate on the values. .NET provides several classes to do this, for example HashTable, ObjectList and Dictionary, but the collection you are most likely to use is an array.

foreach has a variable which is the current item in the enumerator, In the following

foreach (int i in new int[] {1, 2, 3, 5, 8, 13})

```
    Console.Write("{0}, ", i);
```

the collection is created on the fly and is an array of ints; the current item is the variable i, so the output of the code above is

1, 2, 3, 5, 8, 13

Listing 8.28 //each5.cs using System;

```
class ForEachLoop {
    public static void Main() {
        string[] names = {"Parul", "Jayant", "Madhvi", "Rina"};
        foreach (string person in names) {
            Console.WriteLine("{0} ", person);
        }
    }
}
```

The output of this program shall be displayed on console as:

8.15.1 Syntax of foreach-statement:

foreach (type identifier in expression) embedded-statement

The type and identifier of a foreach statement declare the iteration variable of the statement.The iteration variable corresponds to a read-only local variable whose scope extends over the embedded statement. During execution of a foreach statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to assign to the iteration variable or pass the iteration variable as a ref or out parameter.

The type of the expression of a foreach statement must be a collection type (as defined subsequently), and an explicit conversion must exist from the element type of the collection to the type of the iteration variable.

A type C is said to be a collection type if all of the following are true:

1. C contains a public instance method with the signature GetEnumerator() that returns a struct-type, class-type, or interface-type, in the following called E.
2. E contains a public instance method with the signature MoveNext() and the return type bool.
3. E contains a public instance property named Current that permits reading. The type of this property is said to be the element type of the collection type.

The System.Array type is a collection type, and since all array types derive from System.Array, any array type expression is permitted in a foreach statement. For single-dimensional arrays, the foreach statement enumerates the array elements in increasing index order, starting with index 0 and ending with index Length -1. For multi-dimensional arrays, the indices of the rightmost dimension are incremented first.

A foreach statement is executed in the following sequence:

1. The collection expression is evaluated to produce an instance of the collection type. This instance is referred to as c in the following. If c is of a reference-type and has the value null, a NullReferenceException is thrown.
2. An enumerator instance is obtained by evaluating the method invocation c.GetEnumerator(). The returned enumerator is stored in a temporary local variable, in the following referred to as e. It is not possible for the embedded statement to access this temporary variable. If e is of a reference-type and has the value null, a NullReferenceException is thrown.
3. The enumerator is advanced to the next element by evaluating the method invocation e.MoveNext().

If the value returned by e.MoveNext() is true, the following steps are performed:

1. The current enumerator value is obtained by evaluating the property access e.Current, and the value is converted to the type of the iteration variable by an explicit conversion. The resulting value is stored in the iteration variable such that it can be accessed in the embedded statement.
2. Control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a continue statement), another foreach iteration is performed, starting with the step above that advances the enumerator.

3. If the value returned by e.MoveNext() is false, control is transferred to the end point of the foreach statement.

Listing 8.29

```
//foreach3.cs using System;
  using System.Collections;
   class Test {
   public static void Main()
   {
   Hashtable hash = new Hashtable();
   hash.Add("Jyotsna", "Bajaj");
   hash.Add("Ruby", "Sethi");
   hash.Add("Dr..", "kiran sharma");
   hash.Add("Hemlata", "Rajput");
   hash.Add("Sapna", "Jain");
   foreach (string firstName in hash.Keys)
    {
      Console.WriteLine("{0} {1}", firstName, hash[firstName]);
    }
  }
}
```

The output of this program shall be displayed on console as:

```
C:\>each3
Jyotsna Bajaj
Ruby Sethi
Sapna Jain
Dr.. kiran sharma
Hemlata Rajput
```

8.16 Jump Statements

Jump statements are for transferring controls unconditionally :

The following are some of the jump statements:

- break-statement
- continue-statement
- goto-statement
- return-statement
- throw-statement

The location to which a jump statement transfers control is called the target of the jump statement. When a jump statement occurs within a block, and when the target of the jump statement is outside that block, the jump statement is said to exit the block. While a jump statement may transfer control out of a block, it can never transfer control into a block. Execution of jump statements is complicated by the presence of intervening try statements. In the absence of such

try statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening try statements, execution is more complex. If the jump statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement.This process is repeated until the finally blocks of all intervening try statements have been executed.

Listing 8.30

```
static void F() {
  while (true)
{
    try
    {
       try
          {
             Console.WriteLine("Before break");
              break;
           }
            finally
               {
                   Console.WriteLine("Innermost finally block");
               }
          }
          finally
               {
                 Console.WriteLine("Outermost finally block");
               }
     }
     Console.WriteLine("After break");
}
```

the finally blocks associated with two try statements are executed before control is transferred to the target of the jump statement.

8.17 The Break Statement

The break statement exits the nearest enclosing switch, while, do, for, or foreach statement.

```
break-statement:
     break   ;
```

The target of a break statement is the end point of the nearest enclosing switch, while, do, for, or foreach statement. If a break statement is not enclosed by a switch, while, do, for, or foreach statement, a compile-time error occurs. When multiple switch, while, do, for, or foreach statements are nested within each other, a break statement applies only to the innermost

statement. To transfer control across multiple nesting levels, a goto statement (õ8.9.3) must be used. A break statement cannot exit a finally block (õ8.10). When a break statement occurs within a finally block, the target of the break statement must be within the same finally block, or otherwise a compile-time error occurs.
A break statement is executed as follows:

1. If the break statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
2. Control is transferred to the target of the break statement. Since a break statement unconditionally transfers control elsewhere, the end point of a break statement is never reachable.

8.18 The Continue Statement

The continue statement starts a new iteration of the nearest enclosing while, do, for, or foreach statement.

```
continue-statement:
    continue   ;
```

The target of a continue statement is the end point of the embedded statement of the nearest enclosing while, do, for, or foreach statement. If a continue statement is not enclosed by a while, do, for, or foreach statement, a compile-time error occurs. When multiple while, do, for, or foreach statements are nested within one another, a continue statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement must be used. A continue statement cannot exit a finally block. When a continue statement occurs within a finally block, the target of the continue statement must be within the same finally block, or otherwise a compile-time error occurs.
A continue statement is executed as follows:

1. If the continue statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
2. Control is transferred to the target of the continue statement.

As mentioned earlier,since a continue statement unconditionally transfers control elsewhere, the end point of a continue statement is never reachable.

8.19 The Goto Statement

The C# switch statement allows branching to be made on integral and string values. It prohibits the fall through effect of C++ switches, so the following code will not compile:

```
// C++ code
switch (x) { case 1:
   // handle 1
   break;
case 2:
   // handle 2
case 3:
   // handle 2 and 3
   break;
default:
   // handle everything else
}
Instead, C# allows you to go to a branch, explicitly:
// C# code switch (x) { case 1:
   // handle 1
   goto default;
case 2:
   goto case 3;
case 3:
   // handle 2 and 3
   break;
default:
   // handle 1 and everything else
}
```

goto can also be used to make execution flow to a specified label similar to how it is done in C++. The goto statement transfers control to a statement that is marked by a label.

```
goto-statement:
     goto   identifier   ;
     goto   case   constant-expression   ;
     goto   default   ;
```

The target of a goto identifier statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a goto statement to transfer control out of a nested scope, but not into a nested scope. Consider the following Listing:

Listing 8.31

```
class Test {
     static void Main(string[] args) {
          int i = 0;
          while (true) {
               Console.WriteLine(i++);
               if (i == 10)
```

```
                goto done;
            }
        done:
            Console.WriteLine("Done");
        }
    }
```

The output of this program will be displayed on console as:

A goto statement is used to transfer control out of a nested scope. The target of a goto case statement is the statement list of the switch section in the nearest enclosing switch statement that contains a case label with the given constant value. If the goto case statement is not enclosed by a switch statement, if the constant-expression is not implicitly convertible to the governing type of the nearest enclosing switch statement, or if the nearest enclosing switch statement does not contain a case label with the given constant value, a compile-time error occurs.

The target of a goto default statement is the statement list of the switch section in the nearest enclosing switch statement that contains a default label. If the goto default statement is not enclosed by a switch statement, or if the nearest enclosing switch statement does not contain a default label, a compile-time error occurs.

A goto statement cannot exit a finally block. When a goto statement occurs within a finally block, the target of the goto statement must be within the same finally block, or otherwise a compile-time error occurs.

A goto statement is executed as follows:

1. If the goto statement exits one or more try blocks with the associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
2. Control is transferred to the target of the goto statement.

Because a goto statement unconditionally transfers control elsewhere, the end point of a goto statement is never reachable.

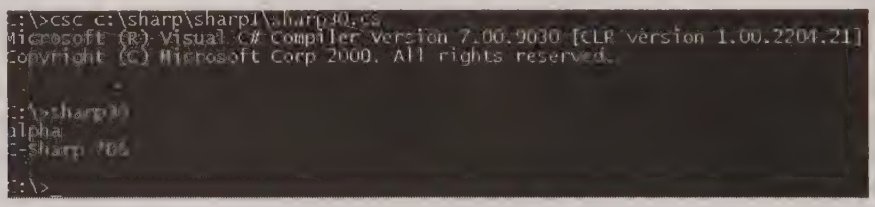
8.20 The Return Statement

The return statement returns control to the caller of the function member in which the return statement appears. Let us see how this is acheived.

Listing 8.32

```
//sharp30.cs
  class xyz30
   {
   static void Main()
   {
     int alpha;
     alpha=beta();                    //line 7
     System.Console.WriteLine("C-Sharp {0}",alpha); //line8
   }
   static int beta()                  //line 10
    {
      System.Console.WriteLine("beta");
      return 786;
     }
}
```

The output of this program shall be displayed on console as:



In program sharp30.cs, return statemnt has been used.The variable alpha has been assigned the value of function beta() i.e. function beta is called upon to get executed. This causes the program to jump to function beta in line 10 which first prints beta and then sets the value of alpha variable to 786 by virtue of return statement. After executing the beta function, the program goes to the next line i.e. line 8 and prints the string C-Sharp first and then value of the variable alpha which is now = 786 next.

Listing 8.33

```
//sharp31.cs
 class xyz31 {
  static void Main()
{
   int alpha;
   alpha=beta();
   System.Console.WriteLine("C-Sharp {0}",alpha);
}
   static void beta()
   {
```

```
        System.Console.WriteLine("beta");
        return 786;
    }
}
```

The output of this program shall be displayed on the console as:

```
C:\>csc \sharp31.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp31.cs(7,10): error CS0022: Cannot implicitly convert type 'void' to 'int'
sharp31.cs(13,6): error CS0127: Since 'xyz31.beta()' returns void, a return keyw
ord must not be followed by an object expression
```

The error message is due to the fact that we have declared the function beta as void, which means there will be no return value for this function. But in the definition of function code we have specified return 786 which is int type whereas we have specified void type return value. Due to this mismatch, the error message has been given. Remember that in program sharp30.cs we have specified int type return value.

Listing 8.34

```
//sharp32.cs
  class xyz32 {
   static void Main()
{
   int alpha;
   alpha=beta();
   System.Console.WriteLine("C-Sharp {0}",alpha);
}
   static int beta()
{
   System.Console.WriteLine("beta");
   return 786;
   System.Console.WriteLine("OK");
} }
```

The output of this program shall be displayed on console as:

```
C:\>csc \sharp32.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp32.cs(14,4): warning CS0162: Unreachable code detected
```

```
C:\>sharp32
beta
C-Sharp 786
```

The program sharp32.cs is a replica of program sharp30.cs except that one more line is added at the end. it may be noted that the program has not executed this code for the reason that we

have included return statement just prior to this line. No code is executed after the return statement. Since the program was unable to execute this statement, the error message has been displayed.

8.20.1 Syntax of return-statement:

return expressionopt ;

A return statement with no expression can be used only in a function member that does not compute a value, that is, a method with the return type void, the set accessor of a property or indexer, a constructor, or a destructor. A return statement with an expression can only be used only in a function member that computes a value, that is, a method with a non-void return type, the get accessor of a property or indexer, or a user-defined operator. An implicit conversion must exist from the type of the expression to the return type of the containing function member. It is an error for a return statement to appear in a finally block. A return statement is executed as follows:

1. If the return statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member by an implicit conversion. The result of the conversion becomes the value returned to the caller.
2. If the return statement is enclosed by one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all enclosing try statements have been executed. * Control is returned to the caller of the containing function member.

Because a return statement unconditionally transfers the control elsewhere, the end point of a return statement is never reachable.

8.21 Common Programming Errors related with control structuring

The following common errors are to be guarded against.

1. The while statement does not include the word do. For this reason, it is a logical error to write

```
while ( condition ) do
action
```

2. The if statement does not include the word then. For this reason, it is an error to write

```
if ( condition ) then
     action
```

3. In the if statement, the condition must be enclosed in parentheses. For this reason, it is an error to write

If condition 4.In the if statement, no semicolon follows the word else. For this reason, it is a logical error to write

```
    if ( condition )
        action 1
else ;
    action 2
```

5. Using a single equals sign = when a double equals sign == is intended is a logical error. It may be syntactically correct to

```
write
 if ( x = 1 )  .  .  . but the way the system interprets this command
is usually not  what the user intended.
                                                    103-4/b/richard
```

6. The break and continue statements affect the innermost loop only. For example, in 1

```
for   ( ... ) 2        for ( ... )
        {
          .........
          .........
      break;
          .........
          .........
      }
```

the break statement causes an exit from for loop 2 but not from for loop 1. 7. In the switch construct, the integer expression that follows switch must be enclosed in parentheses. Thus, it is an error to write

```
    switch i {
......   ......   ......   }
```

8. In the switch construct, the term following case must be a constant expression of integer type (and not a variable). Thus, it is an error to write

```
case i:
when i is a variable.
```

9. Normally, in the switch construct, each group of statements following case or default concludes with break. If break is omitted, execution continues with the statement in the next case or default.

8.22 Summary of C# control statements

Let us summarize what we learnt about the control statements in this chapter:

1. **if-else** : The if-else statement can be sued to select one of two mutually exclusive courses of action in a program. The body of statement following the if statement is executed when the expression is true. When the expression is false and an else is included with the if, the body of statements following the else is executed.

2. **while** : The body of statements following the while loop statement is executed until the conditional expression become false(zero). The body does not execute when the expression is false in its first evaluation.
3. **do-while** : The statements following the do statement are executed and then the expression in the while statement is evaluated. The body of statements following the do is executed until the conditional expression becomes false(zero). The body of statement following the do will always execute at least once.
4. **for** : The body of statements following the for loop statement is executed until the conditional expression becomes false(zero). The body of statements does not execute when the expression is false in its first evaluation.
5. **switch** :The switch statement controls the selection of one body of statements to be executed based upon the value of a case selector expression whose value equals the evaluated expression in the switch statement.
6. **break** : The break statement causes an immediate exit from the control statement in which it appears.
7. **continue** : The continue statement transfers program control to the place in the program such that the next iteration of the loop begins.
8. **goto** : The goto statement transfers control to the C statement that follows the label designated in the goto statement.

Abstract and Base Classes

9.1 Abstract Modifier

The *abstract modifier* can be used with classes, methods, and properties.

We use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes.

Abstract classes have the following features:

1. An abstract class cannot be instantiated.
2. An abstract class may contain abstract methods and accessors.
3. It is not possible to modify an abstract class with the sealed modifier, which means that the class cannot be inherited.
4. A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.
5. Use the abstract modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

1. An abstract method is implicitly a virtual method.
2. Abstract method declarations are only permitted in abstract classes.
3. Since an abstract method declaration provides no actual implementation, there is no method body. The method declaration simply ends with a semicolon and there are no braces ({ })following the signature. For example:

```
public abstract void MyMethod();
```

The implementation is provided by an overriding method, which is a member of a non-abstract class. It is an error to use the static, virtual, or override modifiers in an abstract method declaration. Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

It is an error to use the abstract modifier on a static property. An abstract inherited property can be overridden in a derived class by including a property declaration that uses the override modifier. An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

```
interface I {
   void M();
} abstract class C: I {
   public abstract void M();
} For more information, see 10.1.1.1 Abstract classes and10.5.5
Abstract methods.
```

Example:
In this example, the class MyDerivedC is derived from an abstract class MyBaseC. The abstract class contains an abstract method, MyMethod(), and two abstract properties, GetX() and GetY().

Listing 9.1

```
// Abstract Classes (abstract1.cs)
  using System; abstract class MyBaseC   //Defining Abstract class {
   protected int x = 420;
   protected int y = 786;
   public abstract void MyMethod();   // Abstract method

   public abstract int GetX   // Abstract property
   {
      get;
   }
   public abstract int GetY   // Abstract property
   {
      get;
   }
}
class MyDerivedC: MyBaseC {
   public override void MyMethod()
   {
      x++;
      y++;
   }

   public override int GetX   // overriding property
   {
      get
      {
         return x+10;
      }
```

```
    }
    public override int GetY    // overriding property
    {
        get
        {
            return y+10;
        }
    }
    public static void Main()
    {
        MyDerivedC mC = new MyDerivedC();
        mC.MyMethod();
        Console.WriteLine("x = {0}, y = {1}", mC.GetX, mC.GetY);
    }
}
```

The output of this program shall get displayed as:

```
C:\>abstract1
x = 431, y = 797
```

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

```
MyBaseC mC1 = new MyBaseC();  // Error
```

you will get the following error message:
Cannot create an instance of the abstract class 'MyBaseC'.

9.2 Abstract classes

The *abstract modifier* is used to indicate that a class is incomplete and intended only to be a base class of other classes. An abstract class differs from a non-abstract class in the following ways:

1. An abstract class cannot be instantiated directly, and it is an error to use the new operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be null or contain references to instances of non-abstract classes derived from the abstract types.
2. An abstract class is permitted (but not required) to contain abstract members.
3. An abstract class cannot be sealed.
4. When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract members. Such implementations are provided by overriding the abstract members.

In the example

```
abstract class A {
   public abstract void F();
} abstract class B: A {
   public void G() {}
} class C: B {
   public override void F() {
      // actual implementation of F
   }
}
```

The abstract class A introduces an abstract method F. Class B *introduces an additional method G, but doesn't provide an implementation of F. B must therefore also be declared abstract. Class C overrides F and provides an actual implementation. Since there are no outstanding abstract members in C, C is permitted (but not required)to be non-abstract.

Listing 9.2 : The Engineer Base Class

```
using System;
 class Engineer {
                  // constructor
            public Engineer( string name, float billingRate)
            {
                    this.name = name;
                    this.billingRate =  billingRate;
            }
                    // figure out the charge based on engineer's rate
            public float CalculateCharge(float hours)
            {
                    return(hours * billingRate);
            }
                    // return the name of this type
            public string TypeName()
            {
                    return("Engineer");
            }
            private string name;
            protected float billingRate;
} class Test {
          public static void Main()
          {
                    Engineer engineer = new Engineer("SURI", 21.20F);
                              Console.WriteLine("Name  is:  {0}",
engineer.TypeName());
          }
}
```

9.3 Virtual

The virtual keyword is used to modify a method or property declaration, where the method or the property is called a virtual member. The implementation of a virtual member can be changed by an overriding member in a derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

The virtual modifier with the following modifiers:

*static *abstract *override

Virtual properties behave like abstract methods, except for the differences in declaration and invocation syntax.

It is an error to use the virtual modifier on a static property. A virtual inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

Example:
In this example, the class Dimensions contains the two coordinates x, y, and the Area() virtual method. Different shape classes such as Circle, Cylinder, and Sphere inherit the Dimensions class, and the surface area is calculated for each figure. Each derived class has it own override implementation of Area(). The program calculates and displays the proper area for each figure by invoking the proper implementation of Area() according to the object associated with the method.

Listing 9.3

```
// Virtual1.cs
  using System;
  class TestClass {
   public class Dimensions
   {
      public const double pie = Math.PI;
      protected double x, y;
      public Dimensions()
      {
      }
      public Dimensions (double x, double y)
      {
         this.x = x;
         this.y = y;
      }
      public virtual double Area()
      {
         return x*y;
      }
   }
```

```
    public class Circle: Dimensions
    {
       public Circle(double r): base(r, 0)
       {
       }
       public override double Area()
       {
          return pie * x * x;
       }
    }
    class Sphere: Dimensions
    {
       public Sphere(double r): base(r, 0)
       {
       }
          public override double Area()
       {
          return 4 * pie * x * x;
       }
    }
    class Cylinder: Dimensions
    {
       public Cylinder(double r, double h): base(r, h)
       {
       }
       public override double Area()
       {
          return 2*pie*x*x + 2*pie*x*y;
       }
    }
    public static void Main()
    {
       double r = 4.0, h = 4.0;
       Dimensions c = new Circle(r);
       Dimensions s = new Sphere(r);
       Dimensions l = new Cylinder(r, h);
       // Display results:
       Console.WriteLine("Calculated Area of Circle   = {0}",
                double.Format(c.Area(), "0.00"));
       Console.WriteLine("Calculated Area of Sphere   = {0}",
                double.Format(s.Area(), "0.00"));
       Console.WriteLine("Calculated Area of Cylinder = {0}",
                double.Format(l.Area(), "0.00"));
    }
}
```

The output of this program gets displayed on the console as:

```
C:\>csc \virtual1.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>virtual1
Calculated Area of Circle   = 50.27
Calculated Area of Sphere   = 201.06
Calculated Area of Cylinder = 201.06
```

In the preceding example, notice that the inherited classes Circle, Sphere, and Cylinder are all using constructors that initialize the base class, for example:

```
public Cylinder(double r, double h): base(r, h) {}
```

This is analogous to the C++ initialization list.

9.4 Virtual Methods

When an instance method declaration includes a virtual modifier, the method is said to be a virtual method. When no virtual modifier is present, the method is said to be a non-virtual method.

It is an error for a method declaration that includes the virtual modifier to include the static, abstract, or override modifiers.

The implementation of a non-virtual method is invariant:The implementation is same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be changed by derived classes. The process of changing the implementation of an inherited virtual method is known as overriding the method.

In a virtual method invocation, the run-time type of the instance for which the invocation takes place determines the actual method implementation to invoke. In a non-virtual method invocation, the compile-time type of the instance is the determining factor. In precise terms, when a method named N is invoked with an argument list A on an instance with a compile-time type C and a run-time type R (where R is either C or a class derived from C), the invocation is processed as follows:

First, overload resolution is applied to C, N, and A, to select a specific method M from the set of methods declared in and inherited by C. This is described in.

Then, if M is a non-virtual method, M is invoked. Otherwise, M is a virtual method, and the most derived implementation of M with respect to R is invoked. For every virtual method declared in or inherited by a class, there exists a most derived implementation of the method with respect to that class. The most derived implementation of a virtual method M with respect to a class R is determined as follows:

If R contains the introducing virtual declaration of M, then this is the most derived implementation of M. Otherwise, if R contains an override of M, then this is the most derived implementation of M. Otherwise, the most derived implementation of M is the same as that of the direct base class of R. The following example illustrates the differences between virtual and non-virtual methods:

Listing 9.4

```
class A {
   public void F() { Console.WriteLine("A.F"); }
   public virtual void G() { Console.WriteLine("A.G"); }
} class B: A {
   new public void F() { Console.WriteLine("B.F"); }
   public override void G() { Console.WriteLine("B.G"); }
} class Test {
   static void Main() {
      B b = new B();
      A a = b;
      a.F();
      b.F();
      a.G();
      b.G();
   }
}
```

In the example, A introduces a non-virtual method F and a virtual method G. The class B introduces a new non-virtual method F, thus hiding the inherited F, and also overrides the inherited method G. The example produces the output:

```
A.F B.F B.G B.G
```

Note that the statement a.G() invokes B.G, not A.G. This is because the run-time type of the instance (which is B), not the compile-time type of the instance (which is A), determines the actual method implementation to invoke.

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since most of the derived method are hidden. In the example

Listing 9.5

```
//virtual2.cs
  using System;
  class Alpha {
   public virtual void F() { Console.WriteLine("Alpha.F"); }
} class Beta: Alpha {
   public override void F() { Console.WriteLine("Beta.F"); }
} class Gamma: Beta {
   new public virtual void F() { Console.WriteLine("Gamma.F"); }
} class Delta: Gamma {
   public override void F() { Console.WriteLine("Delta.F"); }
```

```
} class Test {
   static void Main() {
      Delta delta = new Delta();
      Alpha alpha = delta;
      Beta beta = delta;
      Gamma gamma = delta;
      alpha.F();
      beta.F();
      gamma.F();
      delta.F();
   }
}
```

The C and D classes contain two virtual methods with the same signature: The one introduced by A and the one introduced by C. The method introduced by C hides the method inherited from A. Thus, the override declaration in D overrides the method introduced by C, and it is not possible for D to override the method introduced by A. The example produces the following output:

Note that it is possible to invoke the hidden virtual method by accessing an instance of D through a less derived type in which the method is not hidden.

Versioning

This section demonstrates versioning in C# through the use of the override and new keywords. Versioning maintains compatibility between base and derived classes as they evolve.

The C# language is designed such that versioning between base and derived classes in different libraries can evolve and maintain backwards compatibility. It means that the introduction of a new member in a base class with the same name as a member in a derived class is not an error. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that simply hides a similarly named inherited method.

In C#, methods are by default not virtual. To make a method virtual, the virtual modifier has to be used in the method declaration of the base class. The derived class can then override the base virtual method by using the override keyword or hide the virtual method in the base class by using the new keyword. If neither the override keyword nor the new keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class. The following example shows these concepts in action.

Listing 9.6

Example 000: // Versioning\versioning.cs 001: public class MyBase 002:{ 003:public virtual string Meth1() 004: {

```
005:          return "MyBase-Meth1";
006:      }
007:      public virtual string Meth2()
008:      {
009:          return "MyBase-Meth2";
010:      }
011:      public virtual string Meth3()
012:      {
013:          return "MyBase-Meth3";
014:      }
015: }
016:
017: class MyDerived : MyBase
018: {
019:      public override string Meth1()
020:      {
021:          return "MyDerived-Meth1";
022:      }
023:      public new string Meth2()
024:      {
025:          return "MyDerived-Meth2";
026:      }
027:        public string Meth3() // Issues a warning to alert the
programmer
028:                              // that the method hides the inherited
029:                              // member MyBase.Meth3().
030:      {
031:          return "MyDerived-Meth3";
032:      }
033:
034:      public static void Main()
035:      {
036:          MyDerived mD = new MyDerived();
037:          MyBase mB = (MyBase) mD;
038:
039:          System.Console.WriteLine(mB.Meth1());
040:          System.Console.WriteLine(mB.Meth2());
041:          System.Console.WriteLine(mB.Meth3());
042:      }
043: }
```

The output of this program will get displayed as:

```
C:\>csc xml.cs /doc:xml.xml
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>xml
```

Explanation

Line 019 overrides the virtual method Meth1 using the override keyword. Line 023 explicitly hides the virtual method Meth3 using the new keyword. Line 027 hides the virtual method Meth3 because a keyword isn't specified. A warning is issued to alert the programmer that a new or an override keyword should be specified. Hiding a base class member from a derived class isn't an error in C#. This feature enables you to make changes in the base class without breaking other libraries that inherit this base class.

For example, at some point you may have the following classes:

```
class Base {} class Derived: Base {
   public void F() {}
}
```

At some later point, the base class could evolve to add a void method F() as follows:

```
class Base {
   public void F() {}
} class Derived: Base {
   public void F() {}
}
```

Thus, in C#, both the base and derived classes can evolve freely and maintain binary compatibility.

```
override
```

Use the override modifier to modify a method or a property. An override method provides a new implementation of a member inherited from a base class. The method overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method.

You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override.

An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same access level modifier.

You cannot use the following modifiers to modify an override method:

```
new static virtual abstract
```

An overriding property declaration must specify the exact access modifier, type, and name as the inherited property.The overridden property must be virtual, abstract, or override.

Example:

Within the derived class that has an override method, you still can access the overridden base method that has the same name by using the base keyword. For example, if you have a virtual method MyMethod(), and an override method on a derived class, you can access the virtual method from the derived class by using the call:

base.MyMethod() Compare this to the C++ way, where you use the scope resolution operator (::) and the base class name, for example:

```
My_Base_Class_Name::MyMethod()
```

Example:

In this example, there is a base class, Square, and a derived class, Cube. The area of a cube is the sum of the areas of six squares.So it is possible to calculate it by calling the Area() method on the base class.

Listing 9.7

```
// Calling overriden methods from the base class
using System;

class TestClass {
   public class Square
   {
      public double x;
      // Constructor:
      public Square(double x)
      {
         this.x = x;
      }
      public virtual double Area()
      {
         return x*x;
      }
   }
   class Cube: Square
   {
      // Constructor:
      public Cube(double x): base(x)
      {
      }
      // Calling the Area base method:
      public override double Area()
      {
         return (6*(base.Area()));
      }
```

```
    }
    public static void Main()
    {
       double x = 5.2;
       Square s = new Square(x);
       Square c = new Cube(x);
       Console.WriteLine("Area of Square = {0}",
                double.Format(s.Area(), "0.00"));
       Console.WriteLine("Area of Cube = {0}",
                double.Format(c.Area(), "0.00"));
    }
}
```

The output of this program shall get displayed as:

```
Area of Square = 27.04 Area of Cube = 162.24
```

Access modifiers are keywords used to specify the declared accessibility of a member or a type. This section introduces the four access modifiers:

- public
- protected
- internal
- private

The following five accessibility levels can be specified using the access modifiers:

- public
- protected
- internal
- internal
- protected
- private
- public

The public keyword is an access modifier for types and type members. Public access is the most permissive access level. There are no restrictions on accessing public members.
In the following example, two classes are declared, MyClass1 and MyClass2. The public members x and y of the MyClass1 are accessed directly from MyClass2.//Public access (access1)

Listing 9.8

```
using System;
 class TestClass1 {
   public int alpha;
   public int beta;
}
class TestClass2 {
   public static void Main()
   {
      TestClass1 mClass = new TestClass1();
      // Direct access to public members:
```

```
        mClass.alpha = 100;
        mClass.beta = 786;
Console.WriteLine("alpha = {0}, beta = {1}",  mClass.alpha,mClass.beta);
    }
}
```

The output of this program gets displayed on the console as:

If you change the public access level to private or protected, you will get the error message:

'MyClass1.y' is inaccessible due to its protection level.

The protected keyword is a member access modifier. A protected member is accessible from within the class in which it is declared, and from within any class that derived this member.

A protected member of a base class is accessible in a derived class only if the access takes place through the derived class type. For example, consider the following code segment:

Listing 9.9

```
class A {
   protected int x = 123;
}
class B : A {
   void F()
   {
      A a = new A();
      B b = new B();
      a.x = 10;    // Error
      b.x = 10;    // OK
   }
}
```

The statement a.x =10 generates an error because A is not derived from B. Struct members cannot be protected because the struct cannot be inherited.

It is an error to reference a protected member from a class, which is not derived from the protected member's class.

Example:

In this example, the class MyDerivedC is derived from MyClass; therefore, you can access the protected members of the base class directly from the derived class.

Listing 9.10

```
using System;
  class MyClass {
   protected int x;
   protected int y;
}
```

```
class MyDerivedC: MyClass {
   public static void Main()
   {
      MyDerivedC mC = new MyDerivedC();
      // Direct access to protected members:
      mC.x = 10;
      mC.y = 15;
      Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
   }
}
```

Output x = 10, y = 15 If you change the access levels of x and y to private, the compiler will issue the error messages:
'MyClass.y' is inaccessible due to its protection level. 'MyClass.x' is inaccessible due to its protection level.

9.5 Internal

The internal keyword is an access modifier for types and type members. Internal members are accessible only within files in the same assembly.

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide Control and Form classes that cooperate using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a member with internal access outside the assembly within which it was defined.

Example:
This example contains two files, Assembly1.cs and Assembly2.cs. The first file contains an internal base class. In the second file, an attempt to access the member of the base class produces an error.

Listing 9.11 :File Assembly1.cs:

```
// compile with /target:library internal class BaseClass {
   public static int IntM = 0;
} File Assembly2.cs
// compile with /target:exe /reference:Assembly1.dll class TestAccess
{
   public static void Main()
{
      BaseClass myBase = new BaseClass();   // error, BaseClass
not visible outside assembly
   }
}
private
```

The private keyword is a member access modifier. Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared.
Nested types in the same body can also access those private members.

It is a compile-time error to reference a private member outside the class or the struct in which it is declared.

Example:
In this example, the Employee class contains a public member, Name, and a private member, Salary. The public member can be accessed directly, while the private member must be accessed through the public method AccessSalary().

Listing 9.12

```
// Private members
using System;
class Employee {
   public string name = "xx";
   double salary = 100.00;   // private access by default
   public double AccessSalary() {
      return salary;
   }
}
class MainClass {
   public static void Main()
   {
      Employee e = new Employee();
      // Accessing the public field:
      string n = e.name;

      // Accessing the private field:
      double s = e.AccessSalary();
   }
}
```

In the preceding example, if you attempt to access the private members directly, by using a statement like this:

double s = e.salary; you will get the error message:

9.6 Base Classes

A significant part of the power of the .NET framework comes from the base classes supplied by Microsoft as part of the .NET framework. These classes are all callable from C# and provide the kind of basic functionality that is needed by many applications to perform, amongst other things, basic system, Windows, and file handling tasks. To a large extent the .NET base classes can be seen as replacing the previous Win32 API and a large number of MS-supplied COM components.

The base classes are simpler to use, and you can easily derive from them to provide your own specialist classes. You can also use the base classes from any .NET-aware language (calling Win32 API functions from VB was possible but not easy). The types of purposes you can use the base classes to, include:

- String handling
- Arrays, lists, maps etc.
- Accessing files and the file system
- Accessing the registry
- Security
- Windowing
- Windows messages
- Connecting to other computers and to the Internet
- Drawing
- Directory Access
- Database Access

You can see from the above list that besides giving access to basic Windows operations, the base classes define many useful data types, including strings and collections of data.

The base classes are not, of course, only available to C# programs—they can equally well be accessed from VB, C++ or any other .NET-compliant or (by use of some wrapper objects) COM-aware language, but we will concentrate on C# here.

The aim of this chapter is to give you an overview of the kinds of things you can do using the base classes and how to perform certain common operations. Clearly the scope of the base classes is so vast that we cannot give any kind of comprehensive guide in one chapter—instead, we are going to pick on a few common programming tasks and present sample code to demonstrate how you can easily execute those tasks. However, we will also show you how you can use the WinCV tool which is supplied with the .NET SDK to explore the base classes for yourself.

9.6.1 Namespaces for Base Classes

A namespace can be seen as a container for some classes in much the same way that a folder on your file system contains files. Namespaces are needed because there are a lot of .NET classes. Microsoft has written thousands of base classes, and a reasonably large application will define many more. By putting the classes into namespaces we can group related classes together, and also avoid the risk of name collisions: If your company happens to define a class that has the same name as the class written by another organization, and there were no namespaces, there would be no way for a compiler to figure out which class a program is actually referring to. With namespaces, there isn"t a problem because the two classes will be placed in different namespaces, which compares with, say, the Windows files system where files with the same name can be contained in different folders.

It is also possible for namespaces to contain other namespaces, just as folders on your file system can contain other folders as well as files.

When Visual Studio generates your projects, it automatically puts your classes in a namespace. Say for example you use the developer environment to create a C# Windows Application project called MyProject. If you do this and look at the code generated for you, you will see something like this.

```
namespace MyProject{using System; using System.Drawing; using
System.Collections; using System.ComponentModel; using System.WinForms;
using System.Data)
```

There is two C# keywords here that we need to understand: *namespace* and *using*. We will look at namespace first then examine what using does.

The initial namespace command indicates that everything following the opening curly brace is part of a namespace called MyProject. Later on in the file, a class called Form1 is declared.

```
public class Form1 : System.WinForms.Form
```

Since this class has been declared inside the namespace, its "real" name is not Form1 but MyProject.Form1, and any other code outside this namespace must refer to this.This name is correctly known as its fully-qualified name. Note that in the above line, Form1 is derived from a class called Form. This class is defined inside the namespace WinForms, which in turn is defined inside the namespace System (we have already mentioned earlier that it is possible for namespaces to contain other namespaces). This code sample refers to the Form class using its fully qualified name.

Now we will have a look at the purpose of the using command in the above code. It is basically a way of avoiding having to write fully-qualified names everywhere, since the fully-qualified names can get quite long and make your code hard to read. For example consider the line,

```
using System.WinForms;
```

This line declares that we may later in the code use classes from the System.WinForms namespace, without indicating the fully-qualified name—and the same applies to every other namespace mentioned in a using command. For example consider the line of code, also generated by the developer environment

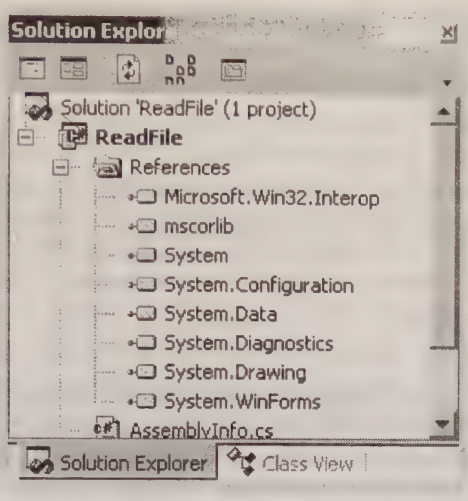
```
public class Form1 : System.WinForms.Form
```

Because of the earlier using command, we could equally well write this as

public class Form1 : Form

In this latter case, the compiler will locate the class by searching all the namespaces that have been mentioned in a using command. If it finds a class named Form in more than one of these namespaces, it will generate a compilation error—in that case, you would need to use the fully-qualified name in your source code.

Note that the only purpose of the using command in this context is to make the code simpler. It doesn't, for example, cause any other code or libraries to be added to your project. The code uses base classes or any other classes that are defined in libraries (recall in .NET these are stored in assemblies), you need to separately ensure that the compiler knows which assemblies to look in for the classes. If you are compiling from the Visual Studio developer environment, this is done through the project references, which are listed in the Solution Explorer window, as shown in the following screenshot:



Some references are inserted automatically when your project is created—which depends on the type of project, and you can add others using the Project. The screenshot shows the situation for a newly created C# Windows Application. (Note that although these are assemblies rather than namespaces that are referenced, the solution explorer shows the namespaces that are found in these assemblies. A given assembly can contain more than one namespace and vice versa.)

If you are compiling from the command line then the assembly mscorlib.dll, which contains some of the most important base classes, is referenced implicitly. You will need to indicate any other assembly which is to be referenced with the command line parameter /r, supplying the full file system path to the assembly. For example, for the above project, the appropriate command is:

```
csc ReadFile.cs  /
r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.Drawing.dll  /
r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.WinForms.dll  /
r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.Data.dll  /
r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.Diagnostics.dll /
r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\System.dll  /
r:c:\WinNT\Microsoft.NET\Framework\v1.0.2204\Microsoft.Win32.Interop.dll
```

Now we have understood the concept of a namespace we can go on to look at the namespaces that contain the various base classes.

If you have a very long namespace and you want to use it several times in your code, then you can substitute a short word for the long namespace name which you can refer to in the code as often as you want. The advantages of doing this are that the code becomes easier to read and maintain and it saves you typing out very long strings.

The syntax for declaring an alias is:

```
using Alias = Prog.SampleCode.CSharpPreview.Examples;
```

and this sample code illustrates how you can use an alias:

Listing 9.13

```
namespace SampleCode.CSharpPreview { using System; using MathEx =
SampleCode.CSharpPreview.Examples; namespace ChBaseClasses { public
class clsExample1 { public static void Main() { MathEx.clsMath MyMathClass
= new MathEx.clsMath(); Console.WriteLine(MyMathClass.Add(3,4)); } } }
// The alias MathEx refers to this namespace.  Examples { public class
clsMath { public int Add(int x, int y) { int z = x + y; return (z); } }
} }
```

As we have remarked there are a huge number of base classes, and there are also a large number of namespaces that contain these classes. We cannot hope to give comprehensive coverage in this chapter, but we will present a summary of some of the more useful namespaces here.

Namespace
Classes Contained

System
These are the most frequently used fundamental classes. For example, the class object, from which every other class is derived, is defined in the System, as are the primitive data types int, string, double etc.

System.Reflection
These are Classes that allow examination of the metadata that describes assemblies, objects and types. Also used for COM Interoperability.

System.IO
Deals with input and output, file handling etc.

System.Collections
These are arrays, lists, linked lists, maps and other data structures.

System.Web
Refers to classes that assist in the generation of web pages using ASP+

System.Net
Contains those classes used to make requests over the network and Internet

System.Data
Are ADO+ classes that make it easy to access data from relational databases and other data sources.

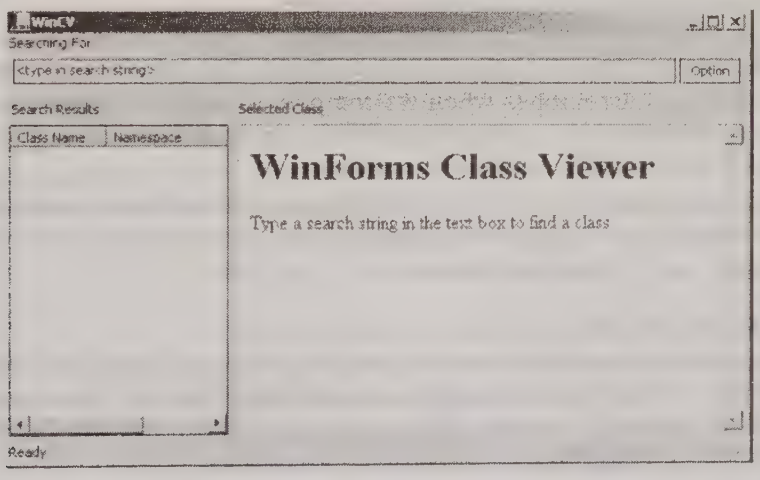
System.WinForms
These are classes to display the controls for standalone (as opposed to web) windows applications.

Microsoft.Win32
Refers to functions that were previously accessible mainly through the Win32 API functions, such as registry access.

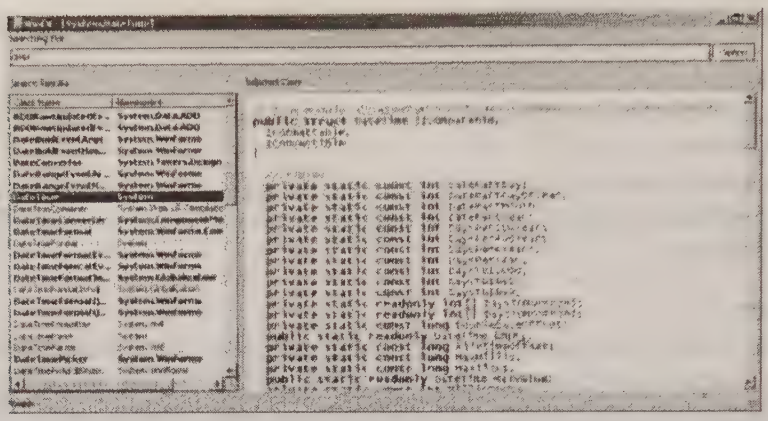
There are many more namespaces, and the best way to learn about them and the classes they contain is to explore them yourself—and Microsoft have provided a tool for just this purpose. We will consider this later on.

The WinCV Tool
WinCV is a class viewer tool, which can be used to examine the classes available in shared assemblies—including all the base classes (A shared assembly is an assembly that has been marked for use by other applications—we will cover shared assemblies in Chapter 10). For each class, it lists the methods, properties, events and fields using a C#-like syntax.

At the present time you start it by typing in WinCV at the command prompt or clicking on the file in Windows Explorer. This gives you a basic window with two panes, as shown.



You then type in a search string that is contained in the class you are looking for in the Searching For box. For example if you want to find a class that handles dates, you might try typing in Date in this box. As you type, WinCV will display a list of classes that contain the string in question in their names:



As this screenshot shows, there are a lot of classes whose name contains the string date—then it is just a question of scanning through the names to find the one that looks most suitable. The definition of the selected class is then displayed in the right hand pane. The left hand pane also indicates the namespace in which the class is contained.

The right hand pane of WinCV tells us which assembly the class is defined in—the screenshot shows us that System.DateTime is defined in mscorlib.dll. This information is useful to indicate if we need to link in any assemblies when we compile, in order to have access to the class. Again, the above screenshot tells us that we are OK for the DateTime class, since it is defined in mscorlib.dll, which is, by default, referenced anyway.

Now we have laid the groundwork, the rest of this chapter is devoted to presenting a number of sample C# projects which are designed to illustrate how to carry out some common programming tasks that are made easy by the base classes.

Manipulating Dates and Times

Our first samples will demonstrate how to access the date-time functionality provided by the base classes. There are two classes of relevance here: DateTime and TimeSpan, both of which are in the System namespace. We start off by displaying the current date and time in various different formats, using this code:

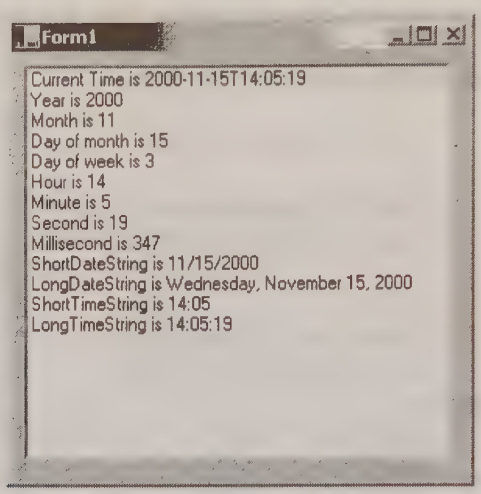
Listing 9.14

```
namespace SampleCode.CSharpPreview.ChBaseClasses
{
      using System;
      using System.Drawing;
      using System.Collections;
      using System.ComponentModel;
      using System.WinForms;
      using System.Data;
      public class FormDisplayDateTimes : System.WinForms.Form
      { ... as before
```

```
                /// <summary>
                /// Summary description for FormDisplayDateTimes.
                /// </summary>
                public FormDisplayDateTimes()
                {
                        // // Required for Win Form Designer support
                        //
                        InitializeComponent();
                        // // TODO: Add any constructor code after
                                //InitializeComponent call
                        //
                        DateTime dtCurrTime = DateTime.Now;
                        AddItem("Current    Time    is    "    +
dtCurrTime.ToString());
                        AddItem("Year is " + dtCurrTime.Year.ToString());
                        AddItem("Month is " + dtCurrTime.Month.ToString());
                        AddItem("Day    of    month    is    "    +
dtCurrTime.Day.ToString());
                        AddItem("Day of week is " +
                                dtCurrTime.DayOfWeek.ToString());
                        AddItem("Hour is " + dtCurrTime.Hour.ToString());
                        AddItem("Minute       is       "       +
dtCurrTime.Minute.ToString());
                        AddItem("Second       is       "       +
dtCurrTime.Second.ToString());
                        AddItem("Millisecond is " +
                                dtCurrTime.Millisecond.ToString());
                        AddItem("ShortDateString is " +
                                dtCurrTime.ToShortDateString());
                        AddItem("LongDateString is " +
                                dtCurrTime.ToLongDateString());
                        AddItem("ShortTimeString is " +
                                dtCurrTime.ToShortTimeString());
                        AddItem("LongTimeString is " +
                                dtCurrTime.ToLongTimeString()); }
```

As explained earlier, we are adding our code to the constructor of the FormDisplayDateTimes class in our project (FormDisplayDateTimes was Form1 in the generated code but we have changed its name to a more meaningful class name.)

We first instantiate a variable dtCurrTime of class DateTime, and initialize it using the static property of the DateTime class, which returns the current date and time. We then use various properties, fields and methods to extract portions of the current date and time. This code produces the following output:



Next we will create another sample, which we will call the FormTimeSpans sample. This shows how to add and take the differences between date-times. This is where the TimeSpan class comes in. The TimeSpan class represents a difference between two date-times. The following sample takes a DateTime, initialized to 1 January 2000, 12 pm, adds an interval of 4 days 2 hours 15 minutes to it, and displays the results along with information about the times and timespan.

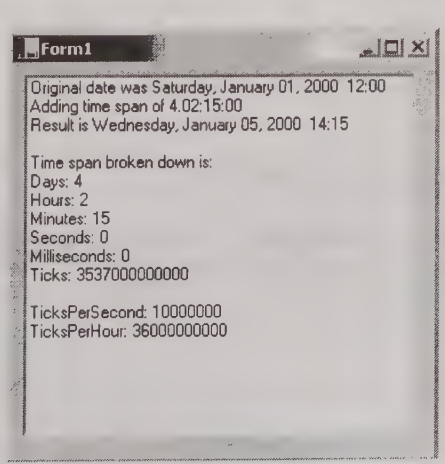
Listing 9.15

```
... public class FormTimeSpans : System.WinForms.Form
{
   ... as before
   public FormTimeSpans()
   {
         // // Required for Win Form Designer support
         // InitializeComponent();
         // // TODO: Add any constructor code after InitializeComponent
call
         //
         // constructor:
         TimeSpan(days, hours, minutes, seconds) TimeSpan Span = new
                  TimeSpan(4,2,15,0);
         // initialize date to 1 Jan 2000, 12 pm
         // constructor: DateTime(year,month,day,hours,minutes,seconds,
         // milliseconds)
         DateTime dtOld = new DateTime(2000,1,1,12,0,0,0);
         DateTime dtNew = dtOld + Span;
```

```
            AddItem("Original date was " + dtOld.ToLongDateString() + " "
+                   dtOld.ToShortTimeString()); AddItem("Adding time
span of " +                 Span.ToString()); AddItem("Result is " +
dtNew.ToLongDateString() + "                          "                 +
dtNew.ToShortTimeString());
            AddItem("");
            AddItem("Time span broken down is:");
            AddItem("Days: " + Span.Days.ToString());
            AddItem("Hours: " + Span.Hours.ToString());
            AddItem("Minutes: " + Span.Minutes.ToString());
            AddItem("Seconds: " + Span.Seconds.ToString());
            AddItem("Milliseconds: " + Span.Milliseconds.ToString());
            AddItem("Ticks: " + Span.Ticks.ToString());
            AddItem(""); AddItem("TicksPerSecond: " +
                    TimeSpan.TicksPerSecond.ToString());
            AddItem("TicksPerHour: " + TimeSpan.TicksPerHour.ToString());
    }
```

In this sample, we see the new operator being used to construct DateTime and TimeSpan instances of given value. Both these classes have several constructors with different numbers of parameters depending on how precisely you wish to specify the time. We add the span on to the time and display the results. In displaying the results we use the ToLongDateString() method of the DateTime class to get the date in plain English, but the ToShortTimeString() method to get the time (using ToLongTimeString() would give us milliseconds as well). We then use various properties of the TimeSpan class to show the days, hours, minutes, seconds, milliseconds and ticks. Ticks are the smallest unit allowed by the TimeSpan class and measure one ten-millionth of a second, as indicated by a couple of static fields that give conversion factors, which we also display.

The above code gives the following output:



The class concepts in itself envisages the idea of deriving new classes from base classes. Let us visualize this with respect to the following set of examples.

The first example is meant for creating the methods of calculating (i.e. making bills for) a member having portfolio as manager.

The class manager will serve as base class for further examples. It has two fields the name as 'private field' and 'billingRate' as protected field.The accessmodifier protected grants the same access as private except that in this case the classes that are derived from this base class also have access to the field. The manager class also has a member function that can be used to calculate the charge based on the nuber of hours of work done.

Listing 9.16

```
//manager1.cs
  using System;
  class Manager {
  // constructor
     public Manager( string name, float billingRate)
     {
         this.name = name;
         this.billingRate =  billingRate;
           }
           //  figure  out the charge based on manager's rate
           public float CalculateCharge(float hours)
           {
                   return(hours * billingRate);
           }
         // return the name of this type
           public string TypeName()
           {
                   return("Manager");
           }
           private  string  name;
           protected float billingRate;
   }
   class Test {
         public static void Main( )
         {
              Manager  manager  =  new  Manager("Hank", 21.20F);
             Console.WriteLine("Name is: {0}", manager.TypeName( ));
         }
}
```

The output of this program shall be:

```
C:\>manager1
Name is:  Manager
```

Now we will derive a class AssistantManager from the above base class Manager.

Listing 9.17

```
//manager2.cs
  using  System;
  class Manager {
          public Manager(string name, float billingRate)
          {
                this.name  =  name;
                this.billingRate  =  billingRate;
           }
           public  float  CalculateCharge(float  hours)
           {
                  return(hours  *  billingRate);
            }
           public  string  TypeName()
           {
                  return("Manager");
           }

           private  string  name;
           protected  float  billingRate;
}
   class  AssistantManager :  Manager {
            public  AssistantManager(string name, float billingRate)

                  base(name, billingRate)
           {
            }
                  // new function, because it's diffrent than the
                  //  base version
            public  new  float  CalculateCharge(float  hours)
            {
                 if  (hours  <  1.0F)
                       hours =  1.0F;          // minimum charge.
                 return(hours  *  billingRate);
             }
                   //  new function, because it's diffrent than the
                   //  base version
             public  new  string  TypeName()
             {
                   return("AssistantManager");
```

```
            }
}
   class  Test {
            public  static  void  Main()
            {
                  Manager m = new Manager("George". 15.50F};
               AssistantManager a = new AssistantManager("Sir John",
40F);
                  Console.WriteLine("{0}  charge  =  {1}",
                                    m.TypeName(),
                                    m.CalculateCharge(2f));
                  Console.WriteLine("{0}  charge  =  {1}",
                                   a.TypeName(),
                                   a.CalculateCharge(0.75F));
            }
}
```

The output of this program is:

```
C:\>manager2
Manager  charge  =  31
AssistantManager  charge  =  40
```

Listing 9.18

```
//manager3.cs
   using  System;
   class  Manager {
        public  Manager(string  name,  float  billingRate)
        {
                this.name  =  name;
                this.billingRate  =  billingRate;
         }
         public float CalculateCharge(float  hours)
         {
                return(hours  *  billingRate);
         }
         public  string  TypeName()
         {
                return("Manager");
          }
                private  string  name;
                protected  float  billingRate;
 }
 class  AssistantManager :  Manager
 {
```

```
        public  AssistantManager(string name, float billingRate)
:
                base(name,  billingRate)
        {
        }
        public new float CalculateCharge(float hours)
        {
                if  (hours  <  1.0F)
                    hours = 1.0F;          //  minimum  charge.
                return(hours  *  billingRate);
        }
        public  new  string  TypeName()
        {
        return("Assistant  Manager");
        }
}
class  Test
{
        public  static  void  Main()
        {
              //  create  an  array  of  managers
         Manager[ ] earray = new Manager[2];
         earray[0] = new Manager("George", 15.50F);
         earray[1] = new AssistantManager("Sir John"), 40F);
        Console.WriteLine("{0} charge = {1}",
                        earray[0].TypeName(),
                        earray[0].CalculateCharge(2F));
        Console.WriteLine("{0} charge = {1}",
                        earray[1].TypeName(),
                        earray[1].CalculateCharge(0.75F));
        }
}
```

The output of this program on the console is as follows:

```
C:\>csc c:\temp\vkj\manager3.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

temp\vkj\manager3.cs(48,75): error CS1002: ; expected
temp\vkj\manager3.cs(48,75): error CS1525: Invalid expression term ','
temp\vkj\manager3.cs(48,78): error CS1002: ; expected
temp\vkj\manager3.cs(48,81): error CS1002: ; expected
temp\vkj\manager3.cs(48,81): error CS1525: Invalid expression term ')'
```

Listing 9.19

```
//manager4.cs
    using  System;
    class  Manager {
          public  Manager(string  name,  float  billingRate)
          {
                this.name =  name;
                this.billingRate  =  billingRate;
           }
                //  function  now  virtual
           virtual public float CalculateCharge(float  hours)
           {
                return(hours  *  billingRate);
            }
                //  function  now  virtual
            virtual  public  string  TypeName()
            {
                return("Manager");
            }
                private  string  name;
                protected  float  billingRate;
            }
            class  AssistantManager:  Manager
            {
                    public  AssistantManager(string  name,  float
billingRate)  :
                      base(name,  billingRate)
                {
                }
                        // overrides function in Manager
                override public float CalculateCharge(float hours)
                  {
                        if  (hours  <  1.0F)
                            hours  =  1.0F;      // minimum charge.
charge.                 return(hours  *  billingRate);
                  }
                        // overrides function in Manager
                      override public string Typename()
                {
                    return("Assistant  Manager");
                 }
            }
            class Test
```

```
        {
                public  static  void  Main()
                {
                        Manager[ ] earray = new Manager[2];
                        earray[0] new Manager("George", 15.50F);
                           earray[1] = new AssistantManager("Sir
                           John", 40F);
                        Console.WriteLine("{0} charge = {1}",
                                          earray[0].TypeName(),
                                      earray[0].CalculateCharge(2F));
                        Console.WriteLine("{0} charge = {1}",
                                          earray[1].TypeName(),
                                   earray[1].CalculateCharge(0.75F));
                 }
         }
```

The output of this program is as follows:

```
C:\>csc c:\temp\vkj\manager4.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

temp\vkj\manager4.cs(37,50): error CS0115: 'AssistantManager.Typename()': no sui
table method found to override
```

Listing 9.20

```
//manager5.cs
    using  System;
    class  Manager {
       public  Manager(string  name,  float  billingRate)
       {
            this.name  =  name;
            this.billingRate  =  billingRate;
        }
        virtual  public  float  CalculateCharge(float  hours)
        {
            return(hours  *  billingRate);
       }
        virtual  public  string  TypeName()
        {
            return("Manager");
        }
         private  string  name;
         protected  float  billingRate;
}
```

```
    class AssistantManager:  Manager {
          public AssistantManager(string name, float billingRate) :
                base(name,  billingRate)
          {
          }
          //  overrides  mistakenly  omitted } class  Test {
        public  static  void  Main()
        {
            Manager[ ]  earray  =  new  Manager[2];
            earray[0] =  new  Manager("George",  15.50F);
            earray[1] = new AssistantManager("Dr.Curie",15.50F
                  Console.WriteLine("{0}  charge  =  {1}",
                                    earray[0].TypeName(),
                                   earray[0].CalculateCharge(2F));
                  Console.WriteLine("{0}  charge  =  {1}",
                                     earray[1].TypeName(),
                                 earray[1].CalculateCharge(0.75F));
        }
}
```

The output of this program on the console is as follows:

```
C:\>manager5
Manager  charge  =  31
Manager  charge  =  34.125
```

Listing 9.21

```
//manager6.cs
using  System;
abstract  class  Manager {
        public  Manager(string  name,  float  billingRate)
        {
                this.name  =  name;
                this.billingRate  =  billingRate;
        }
         virtual  public  float  CalculateCharge(float  hours).
         {
                return(hours  *  billingRate);
         }
         abstract  public  string  TypeName();
         private  string  name;
         protected  float  billingRate;
}
class  AssistantManager: Manager {
         public AssistantManager(string name, float billingRate):
               base(name,  billingRate)
```

```
        {
        }

        override public float CalculateCharge(float hours)
        {
              if (hours < 1.0F)
                  hours = 1.0F;                    // minimum charge.
              return(hours * billingRate);
         }
          override public string TypeName( )
           {
              return("Assistant Manager");
           }
}
class DeputyManager: Manager {
         public DeputytManager(string name, float billingRate) :
              base(name, billingRate)
         {
         }

         override public string TypeName()
         {
              return("Deputy Manager"):
         }
}
class Test {
         public static void Main()
             {
                Manager[ ] earray = new Manager[2];
                   earray[0] = new AssistantManager("Sir John",
40.0F);
               earray[1] = new DeputyManager("Dr. Curie", 45.0F);
                 Console.WriteLine("{0} charge = {1}",
                                        earray[0].TypeName(),
                                    earray[0].CalculateCharge(2F));
                Console.WriteLine("{0} charge = {1}",
                                        earray[1].TypeName(),
                                   earray[1].CalculateCharge(0.75F));
             }
}
```

The output of this program is as follows:

```
C:\>csc c:\temp\vkj\manager6.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

temp\vkj\manager6.cs(44,19): error CS1520: Class, struct, or interface method mu
st have a return type
temp\vkj\manager6.cs(44,70): error CS1002: ; expected
temp\vkj\manager6.cs(45,26): error CS1519: Invalid token ',' in class, struct, o
r interface member declaration
temp\vkj\manager6.cs(45,40): error CS1519: Invalid token ')' in class, struct, o
r interface member declaration
```

9.7 Introduction to the .NET Base Classes

The .NET Base Classes are the means by which you can access much of the core functionality of Windows, as well as perform operations such as data access from .NET code. It consists of a huge number of classes that Microsoft have written to carry out these operations, which you can call in your code. You can even write classes that inherit from the base classes. Here are some of your queries answered:

What is a .NET class?

Assuming that you have some programming experience, you can think of it as follows: For C++ Developers: Think of a .NET class as like a C++ class, except that it's packaged up in a way that means you can easily call it up or inherit from it in any .NET-aware language, including C++, VB and C#.

What Do the Base Classes Do?

The number of base classes is huge, so there's no way that we can cover everything here. But following a quick summary of the kind of things that can be done with them.

1. **Strings**:String handling.
2. **Collections and Arrays**: There are all sorts of permutations for storing sets of objects: Arrays, Lists, Maps, Linked lists. All implemented in highly efficient ways.
3. **WinForms**: Classes to display windows and controls—text boxes, combo boxes, listboxes, file dialogs etc., and to react to events on those controls (Windows messages in low-level C++ parlance).
4. **WebForms**: Classes that do roughly the same as many of the WinForms classes, but are designed to work over the Internet: Instead of creating the relevant control, they generate all the HTML needed to get a web browser to display something that looks and behaves like the control in question. You can use WebForms in your ASP.NET pages.
5. **File Handling**: You can use .NET base classes to navigate around the file system on your machine or network, to check the properties of files, read, modify and write to files, and move and copy files and folders around.
6. **Registry Access**: Base classes are available that allow you to navigate, read, and modify the contents of the registry.

7. **Internet**: Using the base classes you can connect to the Internet and download and upload files.
8. **ADO.NET**: The ADO.NET classes allow you to connect to a data source and read or modify it—much the same abilities as are offered by the existing ADO, except that ADO.NET is designed much more around the concept of disconnected data, and makes heavy use of XML for data transmission.

9.7.1 Collection Classes

This section shows how to implement a collection class that can be used with the foreach statement.

The foreach statement is a convenient way to iterate over the elements of an array.It can also enumerate the elements of a collection,provided that the collection class has implemented the System.Collections.IEnumerator and System.Collections.IEnumerable interfaces.

Example :

The following code sample illustrates how to write a collection class that can be used with foreach. The class is a string tokenizer, similar to the C run-time function strtok.

Listing 9.22

```
// Col.cs //000
using System; //001
using System.Collections;//002
public class Tokens : IEnumerable //003
{ //004
private string[] elements;//005
Tokens(string source, char[] delimiters)//006
{ //007
elements = source.Split(delimiters);//008
} //009
// IEnumerable Interface Implementation //010
public IEnumerator GetEnumerator() //011
{//012
 return new TokenEnumerator(this);//013
} //014
// Inner class implements IEnumerator interface //015
private class TokenEnumerator : IEnumerator //016
{//017
 private int position = -1;//018
 private Tokens t; //019
public TokenEnumerator(Tokens t)//020
{//021
 this.t = t; //022
} //023
public bool MoveNext()//024
```

```
{//025
 if (position < t.elements.Length - 1) //026
{//027
 position++; return true;//028
}//029
else//030
{//031
return false;//032
}//033
} //034
public void Reset() //035
{ //036
position = -1;//037
} //038
public object Current//039
{//040
get//041
{ //042
 return t.elements[position]; //043
}//044
}//045
} //046
// Test Tokens, TokenEnumerator //047
static void Main() //048
{ //049
Tokens f = new Tokens("Nicely prepared program.", new char[]{' ','-
'});//050
 foreach (string item in f)//051
{ //052
Console.WriteLine(item); //053
}//054
} //055
} //056
```

The Output is:

```
Nicely
prepared
program.
```

Explanation

1. The.NET frameworks System namespace is made available on line 001.
2. The.NET frameworks System.Collections namespace is made available on line 002.
3. Line 006 declares the Tokens class.
4. Line 008 uses the String.Split method to parse the string into tokens.

5. Line 011 declares the GetEnumerator method required by IEnumerable. It returns a TokenEnumerator object.
6. Line 020 declares the inner class TokenEnumerator which implements the IEnumerator interface.
7. Line 024 declares the MoveNext method required by IEnumerator.
8. Line 035 declares the Reset method required by IEnumerator.
9. Line 039 declares the Current property required by IEnumerator.
10. Lines 050-056 test Tokens by breaking 'This is a well-done program'.into tokens(using ' 'and '-'as separators) and enumerating those tokens with the foreach statement on line 064.

> Note: Internally, Tokens uses an array, which implements IEnumerator and IEnumerable itself. The code sample could have leveraged the array s enumeration methods as its own, but that would have defeated the purpose of this example.

In C#, it is not strictly necessary for a collection class to inherit from IEnumerable and IEnumerator in order to be compatible with foreach; as long as the class has the required GetEnumerator, MoveNext, Reset, and Current members, it will work with foreach. Omitting the interfaces which have the advantage of allowing you to define the return type of Current to be more specific than object, thereby providing type-safety.

For example, starting with the sample code above, change the following lines:

1. 006: public class Tokens // no longer inherits from IEnumerable
2. 011: public TokenEnumerator GetEnumerator() // doesn't return an IEnumerator
3. 022: public class TokenEnumerator // no longer inherits from IEnumerator * 050: public string Current // type-safe: returns string,not object

Now, because Current returns a string, the compiler can detect when an incompatible type is used in a foreach statement:

```
* 064: foreach(int item in f) // Error: cannot convert string to int
```

The disadvantage of omitting IEnumerable and IEnumerator is that the collection class is no longer interoperable with other .NET languages' foreach statements (or equivalents).
You can have the best of both worlds—type-safety within C# and interoperability with other .NET languages—by inheriting from IEnumerable and IEnumerator and using explicit interface implementation, as demonstrated in the following example.

Example:
This sample is equivalent in function to Example 1, but it provides additional type-safety in C# while maintaining interoperability with other languages.

Listing 9.23

```
// Col2.cs
using System;
using System.Collections;
public class Tokens: IEnumerable {
private string[] elements;
```

```
    Tokens(string source, char[] delimiters)
    {
          elements = source.Split(delimiters);
    }
    // IEnumerable Interface Implementation
    public TokenEnumerator GetEnumerator() // non-IEnumerable version
    {
          return new TokenEnumerator(this);
    }
    IEnumerator IEnumerable.GetEnumerator() // IEnumerable
version
    { return (IEnumerator) new TokenEnumerator(this); }
    // Inner class implements IEnumerator interface
    public class TokenEnumerator: IEnumerator { private int position = -
1;
    private Tokens t;
    public TokenEnumerator(Tokens t)
    {
          this.t = t;
    }
    public bool MoveNext() {
          if (position < t.elements.Length - 1)
          {
                position++;
                return true;
          }
          else
          {
                return false;
          }
    }
    public void Reset()
    {
          position = -1;
    }
    public string Current // non-IEnumerator  version: type-safe
    {
          get {
                return t.elements[position];
          }
          }
          object IEnumerator.Current // IEnumerator  version:
          returns object
          {
                get
                {
```

```
                    return t.elements[position];
                }
            }
        }
    // Test Tokens, TokenEnumerator
    static void Main()
    {
        Tokens f = new Tokens("A nicely prepared  program.", new char []  {'
    ',',-'});
        foreach (string item in f) // try changing string to int
        {
              Console.WriteLine(item);
              }
        }
    }
```

The output of this program shall get displayed as follows:

```
C:\>col2
A
nicely
prepared

program.
```

9.8 Foreach Statement

The foreach statement repeats a group of embedded statements for each element in an array or an object collection. The foreach statement is used to iterate through the collection to get the desired information, but should not be used to change the contents of the collection to avoid unpredictable side effects. The statement takes the following form:

foreach (type identifier in expression) statement where:

type
The type of identifier.

identifier
The iteration variable that represents the collection element.

expression
Object collection or array expression. The type of the collection element must be convertible to the identifier type.

statement
The embedded statement(s) to be executed.

Remarks
The embedded statements continue to execute for each element in the array or collection. After the iteration has been completed for all the elements in the collection, control is transferred to the next statement following the foreach block.

In this section, the following topics are discussed in detail:

1. Using foreach with Arrays
2. Using foreach with Collections

9.8.1 Using foreach with Arrays

When used with an array, the foreach statement repeats the embedded statement(s) for each element in the array.

Example:
In this example, an array of integers is searched for even and odd numbers. A counter for each type of number stores the number of occurrences.

Using foreach with arrays

Listing 9.24

```
col3.cs
using System;
  class MainClass {
   public static void Main()
   {
      int odd = 0, even = 0;
      int[] arr = new int [] {2,9,44,1,7,8,13,25};
      foreach (int i in arr)
      {
         if (i%2 == 0)
            even++;
         else
            odd++;
      }

   Console.WriteLine("Found {0} Odd Numbers, and {1} Even Numbers.",
  odd, even) ;
   }
}
```

The output of this program shall be:

```
C:\>col3
Found 5 Odd Numbers, and 3 Even Numbers.
```

9.8.2 Using foreach with Collections

To iterate through a collection, the collection must meet specific requirements. For example, in the following foreach statement:

foreach (ItemType item in myCollection) myCollection must meet the following requirements:

The collection type:

1. Must be one of the types: interface, class, or struct.
2. Must include an instance method named GetEnumerator that returns a type, for example, Enumerator (explained below).
3. The type Enumerator (a class or struct) must contain:
4. A property named Current that returns ItemType or a type that can be converted to it. The property accessor returns the current element of the collection.
5. A bool method. named MoveNext, that increments the item counter and returns true if there are more items in the collection.

There are three approaches to use collections:

1. Create a collection using the above instructions. This collection can only be used in a C# program.
2. Create a generic collection using the above instructions, in addition to implementing the IEnumerable interface. This collection can be used from other languages such as Visual Basic.
3. Use one of the predefined collections in the collection classes.

The following examples demonstrate the three approaches.

This example Demonstrates Comment

Example 1
This example creates a collection by using the instructions above.

Example 2
This example creates a collection by using the instructions above and implements the IEnumerable and IEnumerator interfaces.

Example 3
Using one of the predefined collection classes. This example creates a Hashtable instance and uses the Hashtable class members to manipulate the collection. The Hashtable class represents a dictionary of associated keys and values, implemented as a hash table.

Listing 9.25: Example 1

```
// Col4.cs
 using System;
// Declare the collection:
   public class MyCollection {
   int[] items;
```

```
    public MyCollection()
    {
       items = new int[5] {125, 440, 380, 342, 750};
    }
    public MyEnumerator GetEnumerator()
    {
       return new MyEnumerator(this);
    }
    // Declare the enumerator class:
    public class MyEnumerator
    {
       int nIndex;
       MyCollection collection;
       public MyEnumerator(MyCollection coll)
       {
          collection = coll;
          nIndex = -1;
       }
       public bool MoveNext()
       {
          nIndex++;
          return(nIndex < collection.items.GetLength(0));
       }
       public int Current
       {
          get
          {
             return(collection.items[nIndex]);
          }
       }
    }
}
public class MainClass {
    public static void Main()
    {
       MyCollection col = new MyCollection();
       Console.WriteLine("Values shown  in this collection are:");
       // Display collection items:
       foreach (int i in col)
       {
          Console.WriteLine(i);
       }
    }
```

The output of this program shall be:

```
C:\>col4
Values shown  in this collection are:
125
440
380
342
750
```

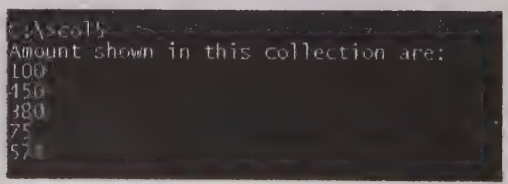
Example 2:
This example repeats the same algorithm of Example 1, but uses a generic collection that can be enumerated from other languages such as Visual Basic. This type of collection must implement the IEnumerable interface from the System.Collections namespace.

Listing 9.26

```
// Col5.cs
using System;
using System.Collections;
//Declare the collection and implement the IEnumerable interface:
public class MyCollection: IEnumerable {
   int[ ] items;
   public MyCollection()
   {
      items = new int[5] {100, 450, 380, 75, 574};
   }
   public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }
   // Implement the GetEnumerator() method:
   IEnumerator IEnumerable.GetEnumerator()
   {
      return GetEnumerator();
   }
   //Declare the enumerator and implement the IEnumerator interface:
   public class MyEnumerator: IEnumerator
   {
      int nIndex;
      MyCollection collection;
      public MyEnumerator(MyCollection coll)
      {
         collection = coll;
         nIndex = -1;
      }
```

```
        public void Reset()
        {
            nIndex = -1;
        }
        public bool MoveNext() {
            nIndex++;
            return(nIndex < collection.items.GetLength(0));
        }
        public int Current
        {
            get
            {
                return(collection.items[nIndex]);
            }
        }
        // The current property on the IEnumerator interface:
        object IEnumerator.Current
        {
            get
            {
                return(Current);
            }
        }
    }
}
public class MainClass {
    public static void Main(string [] args)
    {
        MyCollection col = new MyCollection();
        Console.WriteLine("Amount shown in this collection are:");
        // Display collection items:
        foreach (int i in col)
        {
            Console.WriteLine(i);
        }
    }
}
```

The output of this program will be:



Example 3:

In this example,the predefined Hashtable collection class is used.By using the System.Collections namespace in your program, you can have access to the Hashtable class and its members. To add entries to the Hashtable object, use the Add method.

Listing 9.27

```
Col6.cs // Using the Hashtable collection class
using System;
using System.Collections;
public class MainClass {
   public static void Main(string [] args)
   {
      // Declare a Hashtable object:
      Hashtable ziphash = new Hashtable();
      // Add entries using the Add() method:
      ziphash.Add("11223", "Incroushia");
      ziphash.Add("11552", "Almond");
      ziphash.Add("11201", "Everest");
      ziphash.Add("11101", "Seattle");
      ziphash.Add("11371", "Madagascar");
      // Display contents:
      Console.WriteLine("Zip code\tCity");
      foreach (string zip in ziphash.Keys)
      {
         Console.WriteLine(zip + "\t\t" + ziphash[zip]);
      }
   }
}
```

The output of this program will be as follows:

```
C:\>col6
Zip code        City
11552           Almond
11371           Madagascar
11101           Seattle
11223           Incroushia
11201           Everest
```

10

STRUCTS AND ENUM

10.1 Structs

Structs are similar to classes in that they represent data structures that can contain data members and function members. But unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data—the latter being known as an object.

Structs are particularly useful for small data structures since they have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures are the following:

- they have few data members;
- they do not require use of inheritance or referential identity; and
- they can be conveniently implemented using value semantics where the assignment copies the value instead of the reference.

C# has two varieties that are value types, and these should be familiar to C++ developers: *enum* and *struct*. A struct is essentially a class that is passed by value, they can have the same members as classes and they can implement interfaces. However, you cannot derive from a struct, they cannot have parameterless constructors nor field initializers. Instances of structs are created on the stack and are not garbage collected. Because of this, they are passed by value. All structs are derived from System.Object, so you can call its methods, for example with this struct:

```
struct MyStruct {
   public int i;
}
```

you can use it like this:

```
MyStruct ms; ms.i = 10;
Console.WriteLine("{0}.i = {1}", ms.ToString(), ms.i);
```

This will result in:

```
MyStruct.i = 10
```

Observe the syntax for creating structs: you don't have to use new, because they are created 'inline'. However, if you wish, you can use new and the operator will still create the struct on the stack:

```
int i = 1;
int j = new int();
j = 1;
```

The structs in the table, all of them,implement the IFormattable interface, one method of which is Format(). This overloaded method takes a format string which is used to generate a string. For example:

```
int i = 10; Console.WriteLine("{0}", i.Format("D4", null));
```

will write:

```
00010
```

to the console. The reason for this is that the format type 'D4' means 'four digits, giving 0 when a digit is empty'. This overload takes two parameters, the first is the format string, while the second is a service provider to use to do the formatting. Since we will use Int32, we do not need to provide a service provider and so the special value of null is passed to indicate that no object exists.

There is a marked twist to these base type classes. A literal value is also an object and so we can call methods through that literal.Thus the following statement is perfectly legal:

```
Console.WriteLine("{0}", 45.Format("D4", null));
```

As the name suggests, an enum is an enumerated value, for example:

```
enum Rainbow {Red, Orange, Yellow, Green, Blue, Indigo, Violet}
void Custard() {
   Rainbow colour = Rainbow.Yellow;
   if (colour == Rainbow.Yellow)
      Console.WriteLine("custard colour!");
}
```

Here, we have defined a new type called Rainbow which enumerates all the colours of the rainbow (well, 7 of them really). The Custard() method defines a variable of that type and assigns it a colour. Since the enumerator is a value type you do not have to create it using new. Note that the colour has to be scoped using the enumeration name and the dot operator.

Variables of enumerations can be acted on by the common C# operators, and so in this example, we have used the equality operator == to check if the value is **`Rainbow.Yellow`**. If we want to change this variable to **`Rainbow.Green`** we could use the C# increment operator ++:

```
colour++;
if (colour == Rainbow.Yellow)
   Console.WriteLine("custard colour!");
if (colour == Rainbow.Green)
   Console.WriteLine("lime colour!");
```

The list of similarities between classes and structs is that long—structs can implement interfaces, and can have the same kinds of members as classes. Structs differ from classes in several important ways. However, structs are value types rather than reference types, and inheritance is not supported for structs. Struct values are stored either 'on the stack' or 'in-line'. Also other classes cannot inherit from them. Careful programmers can sometimes enhance performance through judicious use of structs.

The following example shows how structs can be used for encapsulating a point in a graphics system by use of value class.

Listing 10.1: A point struct

```
//structs.cs
using System;
struct Point
{
   public Point(int x, int y)
   {
         this.x = x;
         this.y = y;
   }
   public override string ToString()
   {
         return(String.Format("({0}, {1})", x, y));
   }
   public int x;
   public int y;
}
   class Test
   {
         public static void Main()
         {
               Point   start = new Point(9, 9);
               Console.WriteLine("Start: {0}", start);
         }
   }
```

The output of this program will be displayed on the console as:

```
C:\>struct
Start: (9, 9)
```

In the the main function, a Point is created using the new keyword. For value types, new creates an object on the stack and then calls the appropriate constructor. The x and y coordinates of the point can be thus accessed. Classes are to be used to implement most of the objects, but sometimes it is desirable to create an object that behaves like one of the built-in types; one that is miser and fast to allocate and does not have overheads of references. In such cases, a value type is used, which is done by declaring a struct in C#. For example, the use of a

struct rather than a class for a Point can make a large difference in the number of memory allocations performed by a program. The program below creates and initializes an array of 100 points. With Point implemented as a class, the program instantiates 101 separate objects-one for the array and one each for the 100 elements.

Listing 10.2

```
class Point {
     public int x, y;
     public Point(int x, int y) {
          this.x = x;
          this.y = y;
     }
}
class Test {
     static void Main() {
          Point[] points = new Point[100];
          for (int i = 0; i < 100; i++)
               points[i] = new Point(i, i*i);
     }
}
```

If the Point is instead implemented as a struct, as in

```
struct Point {
     public int x, y;
     public Point(int x, int y) {
          this.x = x;
          this.y = y;
     }
}
```

Then the test program instantiates just one object—the one for the array. The Point instances are allocated in-line within the array. This optimization can be misused. Using structs instead of classes can also make a program slower and fatter, as passing a struct instance as a value parameter causes a copy of the struct to be created. There is no substitute for careful data structure and algorithm design.

10.2 Structs Types

The simple types provided by C#, such as int, double, and bool, are in fact all struct types. Just as these predefined types are structs, so it is possible to use structs and operator overloading to implement new "primitive" types in the C# language.

10.3 Struct Declarations

A struct-declaration is a type-declaration that declares a new struct:

```
struct-declaration:
attributesopt struct-modifiersopt struct identifier
struct-interfacesopt struct-body ;opt
```

A struct-declaration consists of an optional set of attributes followed by an optional set of struct-modifiers, followed by the keyword struct and an identifier that names the struct, followed by an optional struct-interfaces specification, followed by a struct-body, optionally followed by a semicolon.

10.4 Struct Modifiers

A struct-declaration may optionally include a sequence of struct modifiers:

```
struct-modifiers:
struct-modifier
struct-modifiers struct-modifier
struct-modifier:
new public protected internal private
```

It is an error for the same modifier to appear multiple times in a struct declaration. The modifiers of a struct declaration have the same meaning as those of a class declaration Note, however, that the abstract and sealed modifiers are not permitted in a struct declaration. Structs cannot be abstract, and because structs do not permit derivation, they are implicitly sealed.

10.5 Struct Interfaces

A struct declaration may include a struct-interfaces specification, in which case the struct is said to implement the given interface types.

```
struct-interfaces: : interface-type-list
```

- Struct Body

The struct-body of a struct defines the members of the struct.

```
struct-body: { struct-member-declarationsopt }
```

- Struct Members

The members of a struct consist of the members introduced by its struct-member-declarations and the members inherited from the object type.

```
struct-member-declarations: struct-member-declaration
struct-member-declarations   struct-member-declaration
struct-member-declaration: constant-declaration field-declaration
method-declaration property-declaration event-declaration
indexer-declaration operator-declaration constructor-declaration
static-constructor-declaration type-declaration
```

10.6 Enums

An enum type declaration defines a type name for a related group of symbolic constants. Enums are used for 'multiple choice' scenarios, in which a runtime decision is made from a fixed number of choices that are known at compile-time.

Listing 10.3: Enums

```
enum Colour {
     Red,
     Blue,
     Green
}
class Shape {
     public void Fill(Colour colour) {
          switch(colour) {
               case Colour.Red:
                    _
                    break;
               case Colour.Blue:
                    _
                    break;
               case Colour.Green:
                    _
                    break;
               default:
                    break;
          }
     }
}
```

The above program shows a Colour enum and a method that uses this enum. The signature of the Fill method makes it clear that the shape can be filled with one of the given colours.

The use of enums is superior to the use of integer constants—as is common in languages without enums—because the use of enums makes the code more readable and self-documenting. The self-documenting nature of the code also makes it possible for the development tool to assist with code writing and other 'designer' activities. For example, the use of Colour rather than int for a parameter type enables smart code editors to suggest Colour values.

In the example provided in Listing 10.3, a line drawing class uses an enumeration to declare the styles of lines it can draw:

Listing 10.4

```
//line1.cs
using System;
public class Draw {
   public enum LineStyle
   {
     Solid,
```

```
        Dotted,
        DotDash,
    }
     public void DrawLine(int x1, int y1, int x2, int y2, LineStyle
lineStyle)
    {
          switch (lineStyle)
          {
          case LineStyle.Solid:
          // draw solid
          break;
          case LineStyle.DotDash:
          // draw dotdash
          break;

          default:
                throw(new ArgumentException("Invalid line style"));
          }
    }
}
    class Test
    {
      public static void Main()
      {
        Draw draw = new Draw();
       draw.DrawLine(0, 0, 10, 10, Draw.LineStyle.Solid);
       draw.DrawLine(5, 6, 23, 3, (Draw.LineStyle) 35);
      }
}
```

10.7 Enumerator Base Types

The valid base types are byte, sbyte, short, ushort, int, uint, long, and ulong. If the base type is not specified, the default is int. You can specify this as shown in the following coding:

```
enum SmallEnum : byte {
     A, B, C, D
}
```

10.7.1 Initialization

The default value is for first enum member is set to 0, and incremented for each subsequent member. If reqiured, other values may be assigned as shown here:

```
enum SmallEnum : byte {
    A = 1,
    B = 5,
    C = 3,
    D = 42
}
```

You can specify computed values also if they are dependent on values already defined in the enum:

```
enum Values {
   A = 1,
   B = 2,
   C = A + B,
   D = A * C + 33
}
```

If an enum is declared without a 0 value, this may lead to problems, since 0 is the default initialised value for the enum:

```
enum Values {
   A = 1,
   B = 2,
   C = A + B,
   D = A * C + 33
}
   class Test
   {
     public static void Member(Values value)
     {
      // do some processing here
     }
   public static void Main()
   {
     Values value = 0;
     Member(value);
    }
}
```

A member with the value 0 should always be defined as part of an enum.

10.7.2 Enums Declaration

An enum type is a distinct type with named constants. Enum declarations may appear in the same places where class declarations can occur.

The LineStyle enum defines the values that can be specified for the enum, and then that same enum is used in the function call to specify the type of line to draw.

Listing 10.5

```
enum Colour {
      Red,
      Green,
      Blue
}
```

The program in Listing 10.5 declares an enum type named Colour with members Red, Green, and Blue.

An enum declaration declares a new enum type. An enum declaration begins with the keyword enum, and defines the name, accessibility, underlying type, and members of the enum.

enum-declaration:

```
attributesopt enum-modifiersopt enum identifier enum-baseopt enum-body ; opt enum-base: :integral-type enum-body: {enum-member-declarationsopt} {enum-member-declarations,}
```

Each enum type has a corresponding integral type called the 'underlying type of the enum type'. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of byte, sbyte, short, ushort, int, uint, long or ulong. Note that char cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of int.

For example,

```
enum Colour: long {
      Red,
      Green,
      Blue
}
```

declares an enum with an underlying type of long. A developer might choose to use an underlying type of long, as in the aforementioned example, to enable the use of values that are in the range of long but not in the range of int, or to preserve this option for the future.

10.8 Enum Modifiers

An enum-declaration may optionally include a sequence of enum modifiers:

```
enum-modifier enum-modifiers enum-modifier enum-modifier:
new public protected internal private
```

It is an error for the same modifier to appear multiple times in an enum declaration. The modifiers of an enum declaration have the same meaning as those of a class declaration. Note, however, that the abstract and sealed modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

10.9 Enum Members

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

```
enum-member-declarations: enum-member-declaration enum-member-declarations, enum-member-declaration
```

enum-member-declaration:

```
attributesopt identifier attributesopt identifier = constant-
expression
```

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. The example

```
enum Colour: uint {
     Red = -1,
     Green = -2,
     Blue = -3
}
```

is in error because the constant values -1, -2, and -3 are not in the range of the underlying integral type uint. Multiple enum members may share the same associated value. The example

```
enum Colour {
     Red,
     Green,
     Blue,
     Max = Blue }
```

shows an enum that has two enum members—Blue and Max—that have the same associated value. The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a constant-expression initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

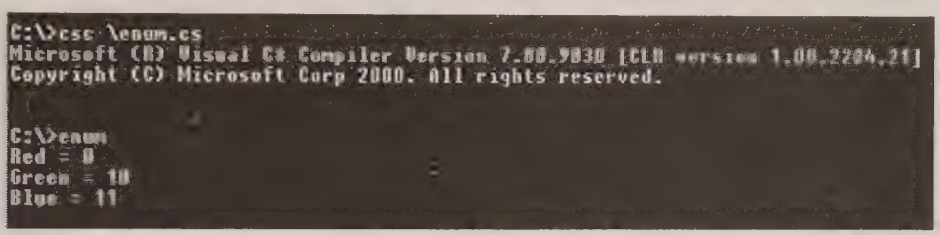
1. If the enum member is the first enum member declared in the enum type, its associated value is zero.
2. Otherwise, the associated value of the enum member is obtained by increasing the associated value of the previous enum member by one. This increased value must be within the range of values that can be represented by the underlying type.

Listing 10.6

```
using System;
enum Colour {
     Red,
     Green = 10,
     Blue
}
class Test {
     static void Main() {
          Console.WriteLine(StringFromColour(Colour.Red));
          Console.WriteLine(StringFromColour(Colour.Green));
          Console.WriteLine(StringFromColour(Colour.Blue));
     }
     static string StringFromColour(Colour c) {
```

```
            switch (c) {
                case Colour.Red:
                    return String.Format("Red = {0}", (int) c);
                case Colour.Green:
                    return String.Format("Green = {0}", (int) c);
                case Colour.Blue:
                    return String.Format("Blue = {0}", (int) c);
                default:
                    return "Invalid colour";
            }
        }
    }
```

The above program prints out the enum member names and their associated values. The output is:


C:\>csc \enum.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>enum
Red = 0
Green = 10
Blue = 11

```
Red = 0 Blue = 11 Green = 10
```

This is explained by the following points:

1. The enum member Red is automatically assigned the value zero (since it has no initializer and is the first enum member);
2. The enum member Green is explicitly given the value 10; and
3. The enum member Blue is automatically assigned the value one greater than the member that textually precedes it.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

```
enum Circular {
    A = B,
    B
}
```

is invalid because the declarations of A and B are circular. A depends on B explicitly, and B depends on A implicitly. Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type.

Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility-an enum member is accessible if its containing enum type is accessible.

10.10 Enum Values and Operations

Each enum type defines a distinct type; an explicit enumeration conversion is required to convert between an enum type and an integral type, or between two enum types. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type, and is a distinct valid value of that enum type. Enum members have the type of their containing enum type (except within other enum member initializers:). The value of an enum member declared in enum type E with associated value v is (E)v. The following operators can be used on values of enum types:

```
==,  !=, <, >, <=, >= , + , - , ^, &, | , ~ , ++, — , sizeof
```

Every enum type automatically derives from the class System.Enum. Thus, inherited methods and properties of this class can be used on values of an enum type.
This program generates Unique Random Numbers and displays on the console.
Listing 10.7

```
usingSpace;
class testrandom {
   public static void Main(string[] args)
{ int intno; if (args.Length == 0)
   {
    Console.WriteLine ("Please enter a parameter eg. unique 5");
    return;
   }
   else
   {
         intno = Int32.Parse (args[0]);
         if (intno < 1)
         {
               // Check to see if user has entered value >= 1 because my
               //  LowerBound is hardcoded to 1
         Console.WriteLine ("Enter value greater than or equal to 1" );
               return;
         }
   }
   unique_random generateit = new unique_random();
  generateit.create_random(intno);
 }
```

```
}

class unique_random
{
   public void create_random (int passed_intno)
   {
         // Remember: C# Requires all variables Initialized.... so lets
         //  initialize variable used
         int LowerBound = 1; int UpperBound = passed_intno;
         bool firsttime = true; int starti = 0;
         int[] vararray; vararray = new int[UpperBound];

         // Random Class used here
         Random randomGenerator = new Random (DateTime.Now.Millisecond);

         do {
               int nogenerated = randomGenerator.Next(LowerBound,
                     UpperBound+1);
               // Note: randomGenerator.Next generates no. to UpperBound
- 1                           //  hence +1
               // .... i got stuck at this pt & had to use the debugger.
               if (firsttime)
               //
               if (firsttime == true)
               {
                     vararray[starti] = nogenerated;
                     // we simply store the  nogenerated in vararray
                     firsttime = false; starti++;
               }
               else //
                     if (firsttime == false)
                     {
                           bool         duplicate_flag      =
CheckDuplicate(nogenerated, starti,  vararray);
                           // call to check in array
               if (!duplicate_flag)
               //
               duplicate_flag == false
               {
                     vararray[starti] = nogenerated; starti++;
               }
         }

   } while (starti < UpperBound);
```

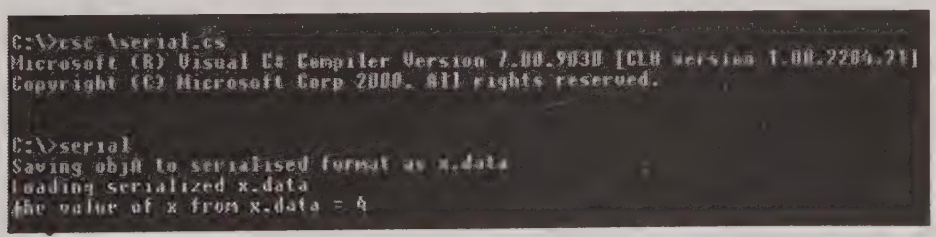
```
    PrintArray (vararray); // Print the array
}
public bool CheckDuplicate (int newrandomNum, int loopcount, int   []
function_array)
{
    bool temp_duplicate = false;

    for (int j = 0; j < loopcount; j++)
    {
          if (function_array[j] == newrandomNum)
          {
                temp_duplicate = true; break;
          }
    }
    return temp_duplicate;
}

// Print Array
public static void PrintArray(Array arr)
{
    Console.Write ("{"); int count = 0; int li = arr.Length;
    foreach (object o in arr)
    {
          Console.Write("{0}", o); count++;
          //Condition to check whether ',' should be added in printing
arrray                if (count < li) Console.Write(", ");
    } Console.WriteLine ("}");
}
}
```

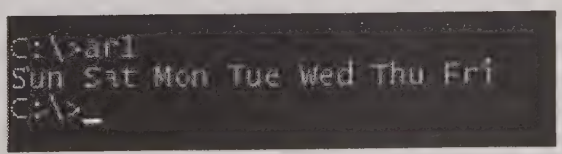
```
C:\>csc c:\temp\vkj\random.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
```

Listing 10.8

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
namespace SG{ public class SGSerialization
{
    private int x;
    public SGSerialization()
    {
          x = 4;
    }
```

```
    public int X
    {
          get { return x; } set { x = value; }
    }
    public static void Main()
    {
          SGSerialization objA = new SGSerialization();
          Console.WriteLine("Saving objA to serialised format as x.data");
          saveProfile("C:\\x.data",objA);
          Console.WriteLine("Loading serialized x.data");
          SGSerialization    objB    =    (SGSerialization)
loadProfile("C:\\x.data");            Console.WriteLine(
"The value of x from x.data = "+objB.X);
    }
    public static void saveProfile(String filePath, Object obj)
    {
          FileStream s = new FileStream(filePath, FileMode.Create,
                FileAccess.Write);
          BinaryFormatter b = new BinaryFormatter();
          b.Serialize(s, obj); s.Close();
    }
    public static Object loadProfile(String filePath)
    {
          FileStream s = new FileStream(filePath, FileMode.Open,
                FileAccess.Read); BinaryFormatter b = new
BinaryFormatter();
          object savedObject = b.Deserialize(s); return (Object)
savedObject;
    }
}
}
```

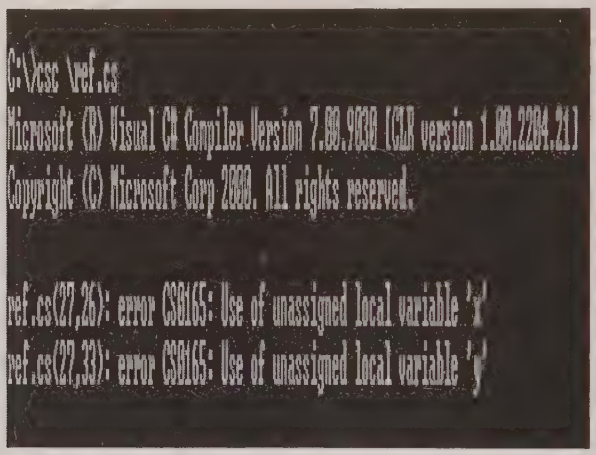

C:\>csc \serial.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>serial
Saving objA to serialised format as x.data
Loading serialized x.data
The value of x from x.data = 4



10.11 Browsing the Internet

Retrieving a file from the Internet really involves two processes: requesting the file, and reading it through a stream. We have already covered the latter process—the code to do it is essentially the same as our earlier sample to read text from a file using the StreamReader class. Loading and requesting the file is a little more complicated as it involves several new classes.

We will need a couple of classes concerned with web browsing: HttpWebRequest, HttpWebResponse and WebRequestFactory, which are all in the System.Net namespace. The assembly that defines this namespace is not by default loaded in C# projects so we need to add it to our references in the solution explorer. Recall that we can do this by right-clicking on the References node in the explorer and selecting Add Reference from the context menu.


C:\>csc \ref.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
ref.cs(27,26): error CS0165: Use of unassigned local variable 'x'
ref.cs(27,33): error CS0165: Use of unassigned local variable 'y'

Next we add a couple of commands to refer to some new namespaces in the C# source file:

Listing 10.9

```
namespace WebRequest { using System; using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.WinForms;
using System.Data;
using System.Net;
using System.IO;
using System.Text;
```

System.Net is for the web classes just mentioned. System.IO is there because we will need to use a StreamReader, and the System.Text provides a helper enumeration value used in constructing the StreamReader. These latter two classes are both in mscorlib.dll so no new references need to be added at compilation time.

Now we can proceed to the code needed to make our request to the web server and display the results.

Listing 10.10

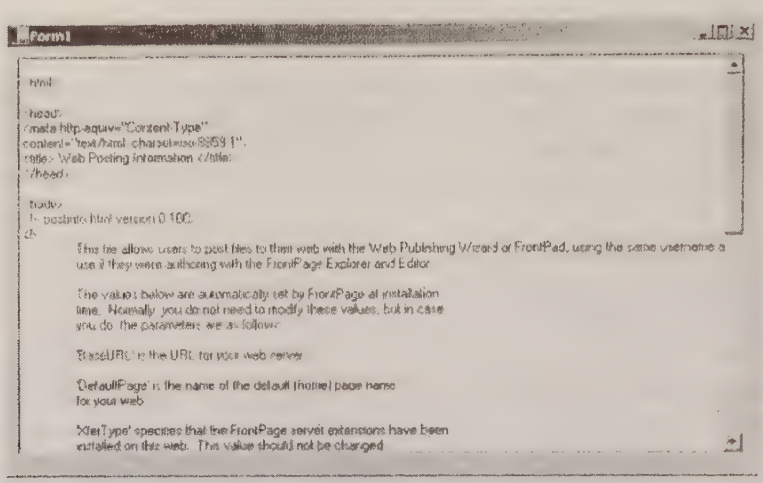
```
HttpWebRequest webreq = (HttpWebRequest)WebRequestFactory.Create ("http:/
/localhost/postinfo.html");
HttpWebResponse webresp = (HttpWebResponse)webreq.GetResponse();
StreamReader strm = new StreamReader( webresp.GetResponseStream(),
Encoding.ASCII);
string sLine;
do {
   sLine = strm.ReadLine();
   AddItem(sLine);
} while (sLine != null);
strm.Close();
```

The request is made through an instance of the class HttpWebRequest. However, it is not possible to construct this instance directly—instead, it needs to be constructed by a static method of another class, WebRequestFactory. The WebRequestFactory.Create() method is designed to create a general request for a given URL— in this case, we pass the URL of one of the files created on the default web site on the local machine when IIS is installed—postinfo.html. Note that WebRequestFactory.Create() actually returns a reference to a WebRequest object, not an HttpWebRequest object, so we need to cast the return value. HttpWebRequest is derived from WebRequest—the latter class is more general-purpose and is able to deal with requests using other protocols besides HTTP.

Once we have the HttpWebRequest instance, we actually make the request by calling its GetResponse() method. GetResponse() returns a WebResponse object, which encapsulates information returned by a web server in response to a request. In a similar manner to WebRequestFactory.Create(), the return value is a reference to a WebResponse rather than an HttpWebResponse, so we need to cast it to the required data type.

Once we have the HttpWebResponse, we simply need to obtain a StreamReader instance that we can use to retrieve the contents of the file. To do this, we use a StreamReaderconstructor that takes two parameters: a more general stream and a value that indicates the encoding type.The stream is obtained from the GetResponseStream() method of the HttpWebResponse class, and the encoding type is Encoding.ASCII, an enumerated value from the System.Text.Encoding class, which indicates that this stream contains ASCII text data.

Although there are a lot of classes involved with this and a lot has to be taken in, the actual code is still reasonably short and simple. Running this sample produces the following result that indicates that the page has been successfully downloaded and displayed:



10.12 Accessing the Registry

In this section, we will present a code sample that enumerates the registry keys and reads their values.

There are several classes used to access the registry, of which we will use two: Registry, and RegistryKey, both of which are in the Microsoft.Win32 namespace. This namespace is defined in the same mscorlib assembly that contains the system namespace, so you have access to it without needing to add further references to your project. The remaining registry classes are concerned with security, which is beyond the scope of this chapter.

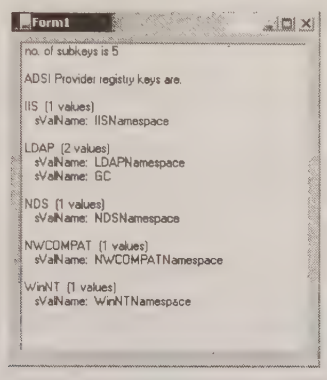
In order to access a given registry key using the base classes it is necessary to progressively navigate down the registry hierarchy to reach the key. We start off using the Registry class: This class contains static fields that allow access to any of the registry hives. The fields are named:

- ClassesRoot
- CurrentConfig
- CurrentUser
- DynData
- LocalMachine
- PerformanceData
- Users

All these are of class RegistryKey. These hives should be familiar to most programmers, though PerformanceData may be unfamiliar to some developers. This hive does exist, and is used to access performance counters, but it is not visible in Regedit. The RegistryKey class represents any given key in the registry and has methods to carry out all sorts of operations including accessing and enumerating subkeys, and reading and modifying values. Thus for example, to access the registry key at the top of the ClassesRoot hive, we would use the following:

```
RegistryKey hkcr = Registry.ClassesRoot;
```

and we would then be able to use the variable hkcr to perform operations on that key.
The sample, RegEnumKeys, binds to a registry key, enumerates its subkeys, and displays the name and all values of each subkey. The key we have chosen to bind to is the registry key whose subkeys contain details of all the ADSI providers installed on the computer, HKLM/SOFTWARE/Microsoft/ADs/Providers. When run, the sample gives this output on the machine:



We first ensure that we can access the registry classes without giving full namespace names:

Listing 10.11

```
namespace Prog.SampleCode.CSharpPreview.ChBaseClasses {
using System;
using Microsoft.Win32;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.WinForms;
using System.Data;
using System.IO;
```

As usual, the action takes place in the constructor to the main form class which we have renamed FormRegEnumKeys. We first need to navigate down to the required key. As mentioned earlier, we cannot do this in one go, but have to progressively step down through the registry hierarchy. Note that the name passed to the RegistryKey.OpenSubKey method is not case-sensitive:

Listing 10.12

```
public FormRegEnumKeys()
{
// // Required for Win Form Designer support
//
InitializeComponent();
//
// TODO: Add any constructor code after InitializeComponent call
//
```

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey software = hklm.OpenSubKey("SOFTWARE");
RegistryKey microsoft = software.OpenSubKey("Microsoft");
RegistryKey ads = microsoft.OpenSubKey("ADS");
RegistryKey prov = ads.OpenSubKey("Providers");
```

Now we can display the number of subkeys(corresponding in this case to the number of installed ADSI providers), and start to enumerate over each of them in a foreach loop. The following code uses two foreach loops—one to enumerate through the subkeys, and the other to enumerate through each value associated with each subkey.

Listing 10.13

```
AddItem("no. of subkeys is " + prov.SubKeyCount);
AddItem("");
AddItem("ADSI Provider registry keys are:");
string [] sProvNames = prov.GetSubKeyNames();
foreach (string sName in sProvNames)
{
   RegistryKey provkey = prov.OpenSubKey(sName); AddItem("");
   AddItem(sName + " (" + provkey.ValueCount.ToString() + " values)" );
   foreach (string sValName in provkey.GetValueNames())
   {
         AddItem(" sValName: " + provkey.GetValue(sValName));
   }
}
```

11

ARRAYS

11.1 Arrays

Array are a data type found in most higher level languages.The array allows the storage of a number of data values in one variable. The underlying concepts of arrays are derived from matrix algebra. Beside being used to represent matrices, arrays are suited for a variety of other purposes because, in C++, this data type represents a defined number of contiguous (i.e. consecutive) bytes in memory.

An array is a data structure that contains a number of variables called the elements of the array. The array elements are accessed through computed indexes. C# arrays are zero indexed; that is, the array indexes start at zero. All the array elements must be of the same type,the type being called the element type of the array. Array elements can be of any type, including an array type. Arrays can be single dimensional or multi-dimensional. Array types are reference types derived from the abstract base type System.Array.

An array makes for the grouping of repetitive data (of the same type) that share the same name. Arrays are useful when dealing with lists of data. The data shares the same name but each element in the array can be accessed individually. An array can be of any data type.

11.2 Concept of Arrays

The values that constitute an array are called its elements.To define an array in C#, the number of elements in the array has to be specified. The elements of an array can be ordered in different pattern schemes. A one-dimensional array is conceptually a single string or a row of a predefined number of values. For example, a linear array of six integers could be considered as:

column	0	1	2	3	4	5
row 0	int1	int2	int3	int4	int5	int6

In array notation, this arrangement will be treated as a 1 x 6 array which means that it consists of a single row and seven columns. In C++, the elements of dimension are numbered with that dimension. An array of more than one dimension is called a multi- dimensional array. For example, a 3 x 6 integer array could be conceptually depicted as shown in Fig.11.1

column	0	1	2	3	4	5
row 0	| int1	| int2	| int3	| int4	| int5	| int6 |
column	0	1	2	3	4	5
row 1	| int7	| int8	| int9	| int10	| int11	| int12|
column	0	1	2	3	4	5
row 2	| int13	| int14	| int15	| int16	| int17	| int18|

Fig. 11.1: An example of a 3 x 6 array

Each of these elements is designated a column number in the range of zero to six and a row number in the range of zero to two. These positional values are known as subscripts or indexes. If the above array is named example then int12 can be identified as example[15]. In this case, the first identification digit inside the square bracket, i.e. 1 specifies that data int12 resides in row 1 and the second digit 5 represents that data int12 resides in the 5th column. Note that the total number of elements is the product of the number of elements in each dimension, i.e. 6 x 3 = 18

An array in C# can thus be differentiated by four attributes:

- Number of dimension (i.e. Nos. of rows and columns);
- Number of elements in each dimensions;
- Data type to the elements;
- Actual value of each element.

The first two attributes determine the 'size' of the array in the conventional mathematical sense, while the next two attributes determine the amount of required memory storage. The elements of an array are accessed by specifying a subscript after the array variable name. The first element of the array has subscript of 0, the next 1, and so on. The last element has a subscript of n-1 where n is the number of elements in the array.

The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

An array has a rank which determines the number of indexes associated with each array element. The rank of an array is also referred to as the dimension of the array. An array with a rank of one is called a single-dimensional array, and an array with a rank greater than one is called a multi-dimensional array.

Each dimension of an array has an associated length which is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the array, but are rather established when an instance of the array type is created at run-time. The length of a dimension determines the valid range of indexes for that dimension: For a dimension of length N, indexes

can range from 0 to N-1 inclusive. The total number of elements in an array is the product of the lengths of each dimension in the array. If one or more of the dimensions of an array have a length of zero, the array is said to be empty.

The element type of an array can be any type, including an array type.

11.3 Single-dimensional Arrays

The following statement declares an array of five integers:

```
int[] myArray = new int [5];
```

This array contains the elements from myArray[0] to myArray[4]. The new operator is used to create the array and initialize the array elements to their default values. In this example, all the array elements are initialized to zero.

An array that stores string elements is declared in the same way. For example:

```
string[] myStringArray = new string[6];
```

11.4 Passing Arrays as Parameters

An initialized array can be passed to a method. For example:

```
PrintArray(myArray);
```

A new array can be initialized and passed in one step with the statement:

```
PrintArray(new int[] {1, 3, 5, 7, 9});
```

In the following example, a string array is initialized and passed as a parameter to the PrintArray method, where its elements are displayed:

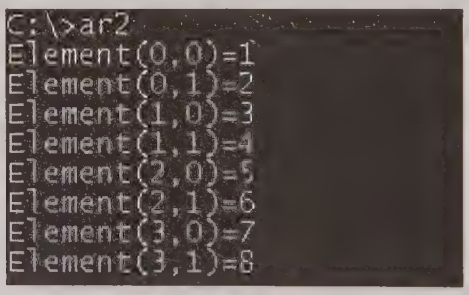
Listing 11.1

```
// Single-dimensional arrays
using System;
public class ArrayClass {
   static void PrintArray(string[] w)
   {
      for (int i=0; i < w.Length; i++)
         Console.Write(w[i] + " ");
   }
   public static void Main()
   {
      // Declare and initialize an array:
      string[] WeekDays = new string []
         {"Sun","Sat","Mon","Tue","Wed","Thu","Fri"};

      // Pass the array as a parameter:
      PrintArray(WeekDays);
   }
}
```

You can pass an initialized array to a method. For example:
PrintArray(myArray); You can also initialize and pass a new array in one step. For example:
Listing 11.2

```
// Two-dimensional arrays
using System;
public class ArrayClass {
   static void PrintArray(int[,] w)
   {
      // Display the array elements:
      for (int i=0; i < 4; i++)
         for (int j=0; j < 2; j++)
            Console.WriteLine("Element({0},{1})={2}", i, j,
w[i,j]);
   }
   public static void Main()
   {
      // Pass the array as a parameter:
      PrintArray(new int[,]{{1,2} , {3,4} , {5,6} , {7,8}});
   }
}
```

The output of this program shall be displayed on console as:



Like all out parameters, an out parameter of an array type must be assigned before it is used; that is, it must be assigned by the callee as shown in the ensuing example:

```
public static void MyMethod(out int[] arr) {
   arr = new int[10];   // definite assignment of arr
}
```

Like all ref parameters, a ref parameter of an array type must be definitely assigned by the caller but, there is no need for it to be definitely assigned by the callee. A ref parameter of an array type may be altered by the call. For example, the array can be assigned the null value or can be initialized to a different array.

Listing 11.3

```
public static void MyMethod(ref int[] arr) {
   arr = new int[10];   // arr initialized to a different array
}
```

The following two examples demonstrate the difference between out and ref when used in passing arrays to methods.

```
C:\>ar1
Sun Sat Mon Tue Wed Thu Fri
C:\>_
```

Listing 11.4 builds an array, myArray, whose elements are arrays. Each one of the array elements has a different size.

Listing 11.4

```
// Array of arrays
 using System;
 public class ArrayTest {
   public static void Main()
   {
      // Declare the array of two elements:
      int[][] myArray = new int[2][];
      // Initialize the elements:
      myArray[0] = new int[5] {1;3,5,7,9};
      myArray[1] = new int[4] {2,4,6,8};
      // Display the array elements:
      for (int i=0; i < 2; i++)
      {
         Console.Write("Element({0}): ", i);
         for (int j=0; j < myArray[i].Length; j++)
            Console.Write("{0} ", myArray[i][j]);

         Console.WriteLine();
      }
   }
}
```

Output

```
Element(0): 1 3 5 7 9 Element(1): 2 4 6 8
```

11.5 Array Initialization

A simple and straightforward method to initialize arrays at declaration-time, is by enclosing the initial values in curly braces ({}). It is important to note that array members are automatically initialized to the default initial value for the array type if not initialized at the time of declaration

The following examples show different methods of initializing different kinds of arrays.

(a) Single-Dimensional Array

```
int[] numbers = new int[5] {1, 2, 3, 4, 5};
string[] names = new string[3] {"Jyoti", "Ruby", "Hema"};
```

You can omit the size of the array, like this:

```
int[] numbers = new int[] {1, 2, 3, 4, 5};
string[] names = new string[] {"Jyoti", "Ruby", "Hema"};
```

You can also omit the new statement if an initializer is provided, like this:

```
int[] numbers = {1, 2, 3, 4, 5};
string[] names = {"Jyoti", "Ruby", "Hema"};
```

(b) Multidimensional Array

```
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
string[,] siblings = new string[2, 2] { {"Mina","Sushma"}, {"Maya","Atul"}
};
```

You can omit the size of the array, like this:

```
int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };
string[,] siblings = new string[,] { {"Mina","Sushma"}, {"Maya","Atul"}
};
```

You can also omit the new statement if an initializer is provided, like this:

```
int[,] numbers = { {1, 2}, {3, 4}, {5, 6} };
string[,] siblings = { {"Mina","Sushma"}, {"Maya","Atul"} };
```

(c) Jagged Array (Array-of-Arrays)

You can also initialize jagged arrays using this style. For example:

```
int[][] numbers = new int[2][] { new int[] {2,3,4}, new int[] {5,6,7,8,9}
};
```

You can also omit the size of the first array, like this:

```
int[][] numbers = new int[][] { new int[] {2,3,4}, new int[] {5,6,7,8,9}
};
```

Example:

The following is a complete C# program that declares and instantiates arrays as discussed above.

Listing 11.5

```
//arrays.cs
using System;
 class DeclareArraysSample
```

```
{
    public static void Main()
    {
        // Single-dimensional array
        int[] numbers = new int[5];

        // Multidimensional array
        string[,] names = new string[5,4];

        // Array-of-arrays (jagged array)
        byte[][] scores = new byte[5][];

       // Create the jagged array
        for (int i = 0; i < scores.Length; i++)
        {
            scores[i] = new byte[i+3];
        }

        // Print length of each row
        for (int i = 0; i < scores.Length; i++)
        {
              Console.WriteLine("Length of row {0} is {1}", i,
scores[i].Length);          }
    }
}
```

Output

Length of row 0 is 3; Length of row 1 is 4; Length of row 2 is 5; Length of row 3 is 6; and Length of row 4 is 7.

Listing 11.6

```
\\array5.cs
 using System;
 class Array {
    public static void Main() {
        int[] myInts = { 7, 14, 21 };
        bool[][] myBools = new bool[2][];
        myBools[0] = new bool[2];
        myBools[1] = new bool[1];
        double[,] myDoubles = new double[2, 2];
        string[] myStrings = new string[3];
       Console.WriteLine("myInts[0]: {0}, myInts[1]: {1},  myInts[2]:
{2}", myInts[0], myInts[1], myInts[2]);
        myBools[0][0] = true;
```

```
            myBools[0][1] = false;
            myBools[1][0] = true;
             Console.WriteLine("myBools[0][0]: {0}, myBools[1][0]:  {1}",
myBools[0][0], myBools[1][0]);
            myDoubles[0, 0] = 3.14756;
            myDoubles[0, 1] = 11.179;
            myDoubles[1, 1] = .00117;
            myDoubles[1, 0] = 65.0012345б;
            Console.WriteLine("myDoubles[0, 0]: {0}, myDoubles[1, 0]:
{1}", myDoubles[0, 0], myDoubles[1, 0]);
            myStrings[0] = "Hem";
            myStrings[1] = "Lata";
            myStrings[2] = "Rajput";
            Console.WriteLine("myStrings[0]: {0}, myStrings[1]: {1},
myStrings[2]: {2}", myStrings[0], myStrings[1], myStrings[2]);
        }
}
```

Listing 11.6 exhibits different implementations of Arrays. The first example is the 'myInts' Array. It is initialized during declaration with explicit values.

Next is a jagged array. It is essentially an array of arrays. We needed to use the 'new' operator to instantiate the size of the primary array and then for each sub-array.

The third example is a two dimensional array. Arrays can be multi-dimensional, with each dimension separated by a comma. It must also be instantiated with the 'new' operator. Finally, we have the one dimensional array of strings.

In each case, you can see that array elements are accessed by identifying the integer index for the item you wish to refer to. Arrays sizes can be any 'int' type value. Their indexes always begin at 0.

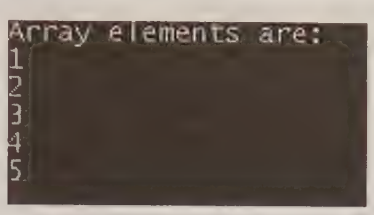
11.5.1 Passing Arrays as Parameters

Listing 11.7

```
using System;
  class TestOut {
   static public void FillArray(out int[] myArray)
   {
      // Initialize the array:
      myArray = new int[5] {1, 2, 3, 4, 5};
   }
   static public void Main()
   {
      int[] myArray; // Initialization is not required
      // Pass the array to the callee using out:
      FillArray(out myArray);
      // Display the array elements:
      Console.WriteLine("Array elements are:");
```

```
        for (int i=0; i < myArray.Length; i++)
            Console.WriteLine(myArray[i]);
    }
}
```

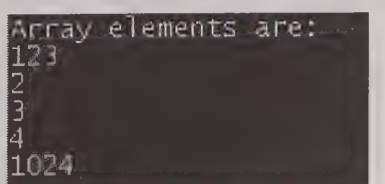
The output of this program shall be displayed on the console as:



Listing 11.8

```
// Using ref to pass the array to a method
using System;
class TestRef {
    public static void FillArray(ref int[] arr)
    {
        // Create the array on demand:
        if (arr == null)
            arr = new int[10];
        // Otherwise fill the array:
        arr[0] = 123;
        arr[4] = 1024;
    }
    static public void Main ()
    {
        // Initialize the array:
        int[] myArray = {1,2,3,4,5};
        // Pass the array using ref:
        FillArray(ref myArray);
        // Display the updated array:
        Console.WriteLine("Array elements are:");
        for (int i = 0; i < myArray.Length; i++)
            Console.WriteLine(myArray[i]);
    }
}
```

The output of this program shall be displayed on the console as:



Array elements are:123 2 3 4 1024

11.6 Array Types Based on Dimensions

We have mentioned that arrays may be single-dimensional or multi-dimensional. Both 'rectangular' and 'jagged' arrays are supported. Let us start with single-dimensional arrays; these being the most common type.Take for example:

Listing 11.9

```
// (array1.cs)
using System;
class Test {
     static void Main() {
          int[] arr = new int[5];
          for (int i = 0; i < arr.Length; i++)
               arr[i] = i * i;
          for (int i = 0; i < arr.Length; i++)
               Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
     }
}
```

This creates a single-dimensional array of int values, initializes the array elements, and then prints out each of them.The program output is:

The type int[] used in the previous example is an array type. Array types are written using a non-array-type followed by one or more rank specifiers. See the example.

Listing 11.10

```
class Test {
     static void Main() {
          int[] a1;          // single-dimensional array of int
          int[,] a2;         // 2-dimensional array of int
          int[,,] a3;        // 3-dimensional array of int
          int[][] j2;        // "jagged" array: array of (array of
int)
          int[][][] j3;      // array of (array of (array of int))
     }
}
```

This shows a variety of local variable declarations that use array types with int as the element type.

Array types are reference types, and so the declaration of an array variable merely sets aside space for the reference to the array. Array instances are actually created via array initializers and array creation expressions.

Listing 11.11

```
class Test {
     static void Main() {
          int[] a1 = new int[] {1, 2, 3};
          int[,] a2 = new int[,] {{1, 2, 3}, {4, 5, 6}};
          int[,,] a3 = new int[10, 20, 30];
          int[][] j2 = new int[3][];
          j2[0] = new int[] {1, 2, 3};
          j2[1] = new int[] {1, 2, 3, 4, 5, 6};
          j2[2] = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9};
     }
}
```

shows a variety of array creation expressions.The variables a1, a2 and a3 denote rectangular arrays, and the variable j2 denotes a jagged array.These terms are based on the shapes of the arrays. Rectangular arrays always have a rectangular shape. Given the length of each dimension of the array, its rectangular shape is evident. For example, the length of a3's three dimensions are 10, 20, and 30 respectively, and it is easy to see that this array contains 10*20*30 elements. In contrast, the variable j2 denotes a 'jagged' array, or an 'array of arrays'. Specifically, j2 denotes an array of an array of int, or a single-dimensional array of type int[]. Each of these int[] variables can be initialized individually, and this allows the array to assume a jagged shape. The example gives each of the int[] arrays a different length. Specifically, the length of j2[0] is 3, the length of j2[1] is 6, and the length of j2[2] is 9.

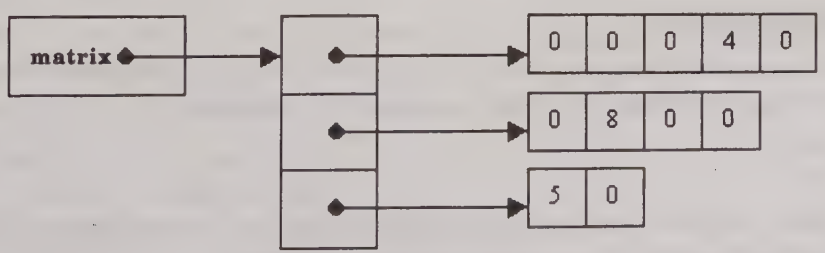


Fig. 11.2: Storage in jagged array

The element type and shape of an array and the number of dimensions it has are parts of its type. On the other hand, the size of the array as represented by the length of each of its dimensions is not part of its type. This split is made clear in the language syntax as the length of each dimension is specified in the array creation expression rather than in the array type. For instance the statement:

```
int[,,] a3 = new int[10, 20, 30];
```

has an array type of int[,,] and an array creation expression of new int[10, 20, 30].
For local variable and field declarations, a shorthand form is permitted so that it is not necessary to re-state the array type.

For instance, the example

```
int[] a1 = new int[] {1, 2, 3};
```

can be shortened to

```
int[] a1 = {1, 2, 3};
```

without any change in program semantics.
The context in which an array initializer such as {1, 2, 3} is used determines the type of the array being initialized. Consider the example.

Listing 11.12

```
class Test {
     static void Main() {
          short[] a = {1, 2, 3};
          int[]   b = {1, 2, 3};
          long[] c = {1, 2, 3};
     }
}
```

It will be seen that the same array initializer syntax can be used for several different array types since context is required to determine the type of an array initializer, it is not possible to use an array initializer in an expression context.

11.7 Declaring Arrays

Array declaration in C# is different from that in C++. The array variable uses commas to determine how many dimensions the array is to have, for example:

```
int[] one;
int[,] two;
```

The first one has a single dimension, while the second one has two dimensions. Since arrays are reference types, you have to use new to create an instance:

```
one = new int[4];
```

this makes the first one reference a single dimensional array with four items indexed from 0 to 3. In the example in foreach showed above, you can provide initial values for array elements:

```
int[,] two = new int[,]{{1,2},{3,4}};
```

In this case, the array has two dimensions of two items each. Array declarations have another format that allows for 'ragged' arrays where the dimensions are not the same. For example:
Listing 11.13

```
int[][] ragged = new int[2][];
ragged[0] = new int[]{1, 2};
ragged[1] = new int[]{3, 4, 5, 6};
foreach (int[] i in ragged)
{
   foreach(int j in i)
      Console.Write("{0} ", j);
   Console.WriteLine();
}
```

Here, the 'ragged' array has two dimensions. The first dimension has two items, while the second dimension has four. The output of this code is:

```
1 2
3 4 5 6
```

C# supports single-dimensional arrays, multidimensional arrays (rectangular arrays), and array-of-arrays (jagged arrays). The syntax for declaring arrays is as follows:
(a) Single-dimensional arrays:

```
int[] numbers;
```

(b) Multidimensional arrays:

```
string[,] names;
```

(c) Array-of-arrays (jagged):

```
byte[][] scores;
```

11.8 Creating Arrays

Declaring arrays (as shown above) does not actually create them. In C#, arrays are objects (discussed later in this tutorial) and must be instantiated. The syntax to create an array is:
(a) Single-dimensional arrays:

```
int[] numbers = new int[5];
```

(b) Multidimensional arrays:

```
string[,] names = new string[5,4];
```

(c) Array-of-arrays (jagged):

```
byte[][] scores = new byte[5][];
for (int x = 0;x < scores.Length; x++)
{
   scores[x] = new byte[4];
}
```

You can also have larger arrays, say, a three-dimensional rectangular array:

```
int[,,] buttons = new int[4,5,3];
```

You can even mix rectangular and jagged arrays. For example, the following code declares a single-dimensional array of three-dimensional arrays of two-dimensional arrays of type int:

```
int[][,,][,] numbers;
```

Array instances are created by array-creation-expressions or by field or local variable declarations that include an array-initializer.

When an array instance is created, the rank and length of each dimension are established and then these remain constant during the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, or to resize its dimensions.

An array instance created by an array-creation-expression is always of an array type. The **System.Array** type is an abstract type that cannot be instantiated. Elements of arrays created by array-creation-expressions are always initialized to their default value.

11.9 Accessing Array Members

Array members in C# are accessed much like in C or C++. For example, the following code creates an array called numbers and then assigns a 5 to the fifth element of the array:

Listing 11.14

```
int[] numbers = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
numbers[4] = 5;
//The following code declares a multidimensional
//array and assigns 5 to the member located at [1, 1]:
int[,] numbers = { {1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10} };
numbers[1, 1] = 5;
//With jagged arrays, accessing a member is done as follows:
int[][] numbers = new int[][] {
   new int[] {1, 2},
   new int[] {3, 4}
}; numbers[1][1] = 5;
```

Array elements are accessed using element-access expressions of the form A[I1, I2, ..., IN], where A is an expression of an array type and each IX is an expression of type int, uint, long, ulong, or of a type that can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices. The elements of an array may be enumerated using a foreach statement.

11.10 Arrays are Objects

In C#, arrays are actually objects. **System.Array** is the abstract base type of all array types. You can use the properties, and other class members, that **System.Array** has. For example, the Length property can be used to get the length of an array. The following code assigns the length of the numbers array, which is 5, to a variable called LengthOfNumbers:

```
int[] numbers = {1, 2, 3, 4, 5};
int LengthOfNumbers = numbers.Length;
```

The System.Array class provides many other useful methods/properties, such as the methods for sorting, searching, and copying arrays.

11.11 Using foreach on Arrays

C# also provides the **foreach** statement. This statement provides a simple, clean way to iterate through the elements of an array. For example, the following code creates an array called numbers and iterates through it with the foreach statement:

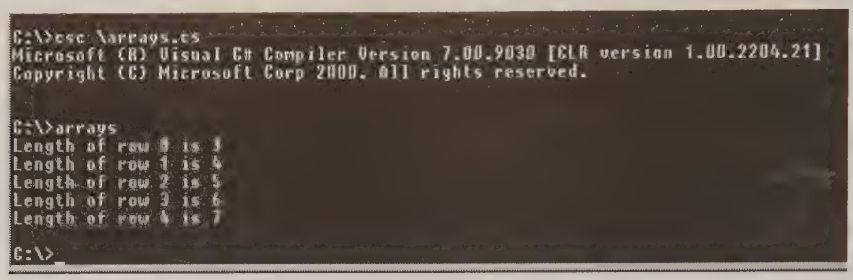
Listing 11.15

```
int[] numbers = {4, 5, 6, 1, 2, 3, -2, -1, 0};
foreach (int i in numbers) {
   System.Console.WriteLine(i);
}
```

Listing 11.16

```
//arrays.cs
using System;
class DeclareArraysSample
{
   public static void Main()
   {
         // Single-dimensional array
         int[] numbers = new int[5];
         // Multidimensional array
         string[,] names = new string[5,4];
         // Array-of-arrays (jagged array)
         byte[][] scores = new byte[5][];
         // Create the jagged array
         for (int i = 0; i < scores.Length; i++)
         {
               scores[i] = new byte[i+3];
         }
         // Print length of each row
         for (int i = 0; i < scores.Length; i++)
         {
               Console.WriteLine("Length of row {0} is {1}", i,
scores[i].Length);
         }
   }
}
```

The output of this program will be displayed on the console as:


C:\>csc \arrays.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>arrays
Length of row 0 is 3
Length of row 1 is 4
Length of row 2 is 5
Length of row 3 is 6
Length of row 4 is 7
C:\>

11.12 Definite Assignment and Arrays

When it comes to definite assignment, arrays work a bit differently. For arrays of both reference and value type (as in classes and structs respectively), an element of an array can be accessed, even though it may not have been initialised.

Listing 11.17

```
 //Define3.cs
using System;
struct Complex {
   public Complex(float real, float imaginary)
{
   this.real = real;
   this.imaginary = imaginary;
}
   public override string ToString()
{
   return(String.Format("({0}, {0})", real, imaginary));
}
   public float real;
   public float imaginary;
}
   class Test
{
   public static void Main()
{
   Complex[] arr = new Complex[10];
   Console.WriteLine("Element 5: {0}", arr[5]);
}
}
```

The output of this program will be displayed on the console as:

```
C:\>csc c:\temp\vkj\define3.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
```

11.13 Array Types

An array type is written as a non-array-type followed by one or more rank-specifiers:

```
array-type: non-array-type  rank-specifiers
non-array-type: type
rank-specifiers: rank-specifier rank-specifiers rank-specifier rank-
specifier: [   dim-separatorsopt   ]
dim-separators:
, dim-separators   ,
```

A non-array-type is any type that is not itself an array-type. The rank of an array type is given by the leftmost rank-specifier in the array-type: A rank-specifier indicates that the array is an array with a rank of one plus the number of "," tokens in the rank-specifier. The element type of an array type is the type that results from deleting the leftmost rank-specifier; rather, an array type of the form T[R] is an array with rank R and a non-array element type T. An array type of the form T[R][R1]_[RN] is an array with rank R and an element type T[R1]_[RN].

In effect, the rank-specifiers are read from left to right before the final non-array element type. For example, the type int[][,,][,] is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of int.

As mentioned earlier, arrays with a rank of one are called single-dimensional arrays. Arrays with a rank greater than one are called multi-dimensional arrays, and are also referred to as two-dimensional arrays, three-dimensional arrays, and so on.

At run-time, the value of an array type can be null or a reference to an instance of that array type.

11.14 The System.Array Type

The System.Array type is the abstract base type of all array types. An implicit reference conversion exists from any array type to System.Array, and an explicit reference conversion exists from System.Array to any array type. Note that System.Array is itself not an array-type. Rather, it is a class- type from which all array-types are derived.

At run-time, a value of type System.Array can be null or a reference to an instance of any array type.

11.15 Array Members

Every array type inherits the members declared by the System.Array type.

11.16 Array Covariance

For any two reference—types A and B—if an implicit reference conversion or explicit reference conversion exists from A to B, then the same reference conversion also exists from the array type A[R] to the array type B[R], where R is any given rank-specifier (but the same for both array types). This relationship is known as array covariance. Array covariance in particular means that a value of an array type A[R] may actually be a reference to an instance of an array type B[R], provided an implicit reference conversion exists from B to A.

Because of array covariance, assignments to elements of reference type arrays include a run-time check which ensures that the value being assigned to the array element is actually of a permitted type. See the example.

Listing 11.18

```
class Test {
     static void Fill(object[] array, int index, int count,
     object value) {
          for (int i = index; i < index + count; i++) array[i] =
    value;
     }
     static void Main() {
          string[] strings = new string[100];
          Fill(strings, 0, 100, "Undefined");
          Fill(strings, 0, 10, null);
          Fill(strings, 90, 10, 0);
     }
}
```

The assignment to array[i] in the Fill method implicitly includes a run-time check which ensures that the object referenced by value is either null or an instance of a type that is compatible with the actual element type of array. In Main, the first two invocations of Fill succeed, but the third invocation causes an ArrayTypeMismatchException to be thrown upon executing the first assignment to array[i]. This occurs because a boxed int cannot be stored in a string array.

Array covariance specifically does not extend to arrays of value-types. For example, no conversion exists that permits an int[] to be treated as an object[].

11.17 Array Initializers Revisited

Array initializers may be specified in field declarations, local variable declarations and array creation expressions: Look at the array-initializer:

Listing 11.19

```
{   variable-initializer-listopt   }
{   variable-initializer-list   ,   }
variable-initializer-list:
variable-initializer variable-initializer-list   ,   variable-initializer
variable-initializer:
expression array-initializer
```

An array initializer consists of a sequence of variable initializers, enclosed by "{"and "}" tokens and separated by "," tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer. The context in which an array initializer is used determines the type of the array being initialized. In an array creation expression, the array type immediately precedes the initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, such as:

```
int[] a = {0, 2, 4, 6, 8};
```

it is simply a shorthand for an equivalent array creation expression:

```
int[] a = new int[] {0, 2, 4, 6, 8}
```

For a single-dimensional array,the initializer must consist of a sequence of expressions which are assignment compatible with the element type of the array. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of expressions in the array initializer determines the length of the array instance being created. For example, the array initializer above creates an int[] instance of length 5 and then initializes the instance with the following values:

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

The array initializer of a multi-dimensional array must have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements must be the same as for the other array initializers at the same level. The example:

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension :

```
int[,] b = new int[5, 2];
```

and then initializes the array instance with the following values:

```
b[0, 0] = 0; b[0, 1] = 1; b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5; b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

When an array creation expression includes an explicit dimension lengths as well as an array initializer, the lengths must be constant expressions and the number of elements at each nesting level must match the corresponding dimension length. Some examples follow:

```
int i = 3;
int[] x = new int[3] {0, 1, 2};      // OK
int[] y = new int[i] {0, 1, 2};      // Error, i not a constant
int[] z = new int[3] {0, 1, 2, 3};   // Error, length/initializer mismatch
```

Here, the initializer for y is in error because the dimension length expression is not a constant, and the initializer for z is in error because the length and the number of elements in the initializer do not agree.

11.18 ArrayList

ArrayList class comes handy in binary searches.

The ArrayList class Implements a collection of objects using an array whose size is dynamically increased as required. The ArrayList class is very useful if you need to add, remove, index, or search data in a collection.

11.18.1 Adding data to an ArrayList

Adding data to an ArrayList is quite easy. Just call Add member of the class:

```
ArrayList dataArray = new ArrayList();
for ( int i = 0; i<20; i++ ) {
    dataArray.Add( i );
} or:
    dataArray.Add( "Object type" );
```

The Add member takes an argument of Object type. You can add any value to the array.

11.18.2 Delete data from an ArrayList

The Remove and RemoveAt members are used to delete data from an ArrayList

```
// Remove the element containing "3".
myAL.Remove( "3" );
```

11.18.3 Searching elements in ArrayList

The BinarySearch member of the ArrayList is useful in this context. This member returns the zero-based index of a specific element in the sorted ArrayList or a portion of it, using a binary search algorithm.

```
int myIndex=myList.BinarySearch( 3 );
```

11.18.4 Difference in handling Arraytypes

Perhaps the biggest difference between C++ and C#, is the way in which they handle arrays.

C# arrays are managed types, meaning that they hold references, not values, and they're garbage collected. You can declare arrays in several ways, including as multidimensional (rectangular) arrays and as arrays of arrays (jagged). Note in the following examples that the square brackets come after the type, not after the identifier as in some languages.

```
int[ ] intArray;                  // A simple array
int[ , , ] intArray;              // A multidimensional array
                                  // of rank 3 (3 dimensions)
int[ ][ ] intArray                // A jagged array of arrays
int[ ][ , , ][ , ] intArray;      // A single-dimensional array
                                  // of three-dimensional arrays
                                  // of two-dimensional arrays
```

Arrays being objects; do not have a size when declared first. For this reason, you must create them after you declare them. Suppose you want an array of size 5. This code will serve the purpose:

```
int[] intArray = new int[5];
```

If you do this twice, it automatically reallocates the array. Therefore

```
int[] intArray; intArray = new int[5];
intArray = new int[10];
```

results in an array called intArray, which has 10 members. Instantiating a rectangular array is similarly done:

```
int[] intArray = new int[3,4];
```

However, instantiating a jagged array is a bit more involved. You might expect to say new int[3][4], whereas you really need to say:

```
int[][] intArray = new int[3][];
for (int a = 0; a < intArray.Length; a++)
{
    intArray[a] = new intArray[4];
}
```

You can initialize a statement in the same line you create and instantiate it by using curly brackets:

```
int[] intArray = new int[5] {1, 2, 3, 4, 5};
```

You can do the same thing with a string-based array:

```
string[] strArray = new string[3] {"MSJ", "MIND","MSDNMag"};
```

If you mix brackets, you can initialize a multidimensional array:

```
int[,] intArray = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
```

You can also initialize a jagged array:

```
int[][] intArray = new int[][] {
new int[] {2,3,4}, new int[] {5,6,7}
};
```

If you leave out the new operator, you can even initialize an array with implicit dimensions:

```
int[] intArray = {1, 2, 3, 4, 5};
```

Arrays are considered objects in C#, and as such they are handled like objects, not like an addressable stream of bytes. Specifically, arrays are automatically garbage collected, so you don't need to destroy them when you're finished using them. Arrays are based on the C# class System.Array, so you can treat them conceptually like a collection object, using their Length property and looping through each item in the array. If you define intArray as shown earlier, the call

```
intArray.Length
```

would return 5. The System.Array class also provides means to copy, sort, and search arrays. C# provides a foreach operator, which operates like its counterpart in Visual Basic, for looping through an array. Consider the code:

```
int[] intArray = {2, 4, 6, 8, 10, -2, -3, -4, 8};
foreach (int i in intArray) {
   System.Console.WriteLine(i);
}
```

This code will print each number in intArray on its own line of the system console. The System.Array class also provides a GetLength member function, so the preceding code could also be written like this (remember, arrays are zero-based in C#):

Listing 11.20

```
for (int i = 0; i < intArray.GetLength(); i++) {
    System.Console.WriteLine(i);
}
//sharp63.cs
  class hema {
   public static void Main()
   {
    int[] alpha;
    alpha = new int[3];
    alpha[0]=10; alpha[1]= 100; alpha[2]=200;    //line 8
    System.Console.WriteLine(alpha[0]);
    alpha[0]++;                                  //line 10
    System.Console.WriteLine(alpha[0]);
    int i;                                       //line 12
    i=1;
    System.Console.WriteLine(alpha[i]);
    i=2;
    System.Console.WriteLine(alpha[i]);
  }
}
```

The output of this program shall be displayed on console as:

```
C:\>csc c:\sharp\sharp1\sharp63.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>sharp63
10
11
100
200

C:\>
```

The above program illustrates a simple array creation. Here the array alpha is of int[] type meaning that it will store integer values only. The statement alpha = new int[3];indicates that

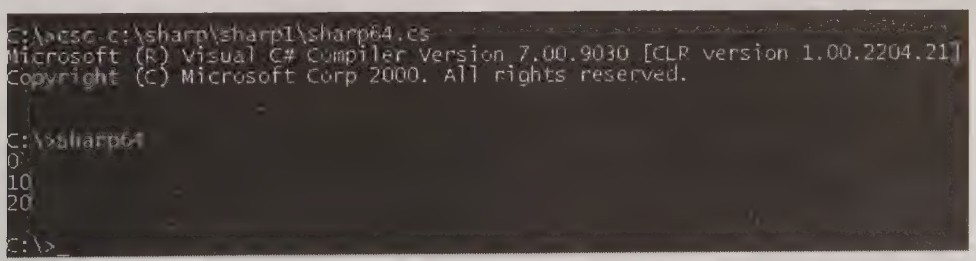
the array is to hold only three values. The three array members are labelled as alpha[0], alpha[1] and alpha[2] in line 11. In programming languages the number system always starts from 0, i.e. the first number is 0, second is 2 and so on. In line number 8, we have setup the value of array-member alpha[0] as 10 and in line number we are asking the program to output its value on console which will cuase computer to print the result as 10.

In line 10, we are asking to put an increment (by default 1) to the content of array-member alpha[0] which at present is 10. So the value printed shall be 11. In line 12 a variable has been defined and initialised to 1 in line 13 and by virtue of line 14 we are printing the value of array-member alpha[i] i.e. alpha[1] which is 100.

Listing 11.21

```
//sharp64.cs
class hema {
   public static void Main()
   {
    int[] alpha;
    alpha = new int[4];
    int i;
    for (i = 0; i<=3; i++)
    alpha[i] = i * 10;
    for(i = 0; i <= 3; i++)
    System.Console.WriteLine(a[i]);
   }
}
```

The output of this program will be displayed on the console as:



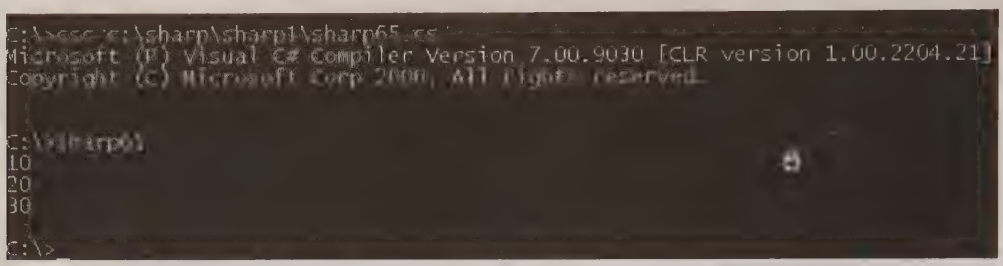
The program sharp64.cs uses two **for** statements, one for in inputting the data in the array alpha and another outputing the data values stored in array alpha.

Listing 11.22

```
//sharp65.cs class hema
  {
   public static void Main()
   {
     int[] alpha;
     alpha = new int[3];
```

```
        alpha[0] = 10 ;
        alpha[1] = 20;
        alpha[2] = 30;
        foreach (int i in alpha)
        System.Console.WriteLine(i);
      }
  }
```

The output of this program will be displayed on the console as:



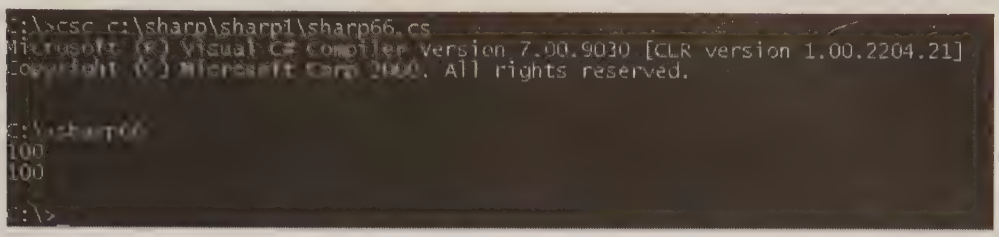
The above program uses foreach statement to the same effect as in the previous example.

11.19 out and ref

Listing 11.23

```
//sharp66.cs
  class hema {
   public static void Main()
   {
    int i=100;
    yyy a;
    a = new yyy();   //line 8
    a.alpha(i);
    System.Console.WriteLine(i);
   }
}
   class yyy
   {
     public void alpha(int i)
     {
       System.Console.WriteLine(i);
       i=10;
     }
   }
```

The output of this program will be displayed on the console as:

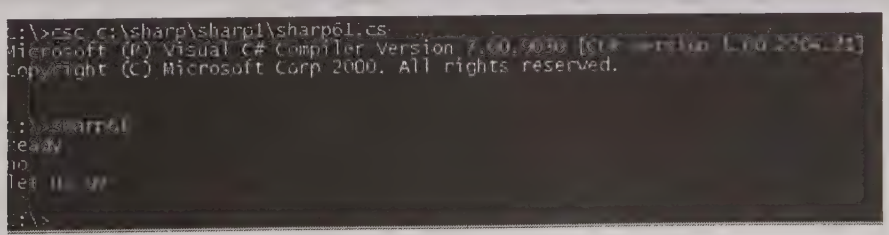


In this example, **a** is an object and variable **i** is initialised to a value 100. Function alpha() is called upon with one parameter i (basically a non-syntactical mistake). We want to print the value of i in function alpha(). When function ablpha() is called upon from Main i had a value of 100. The writeLine will cause the computer to display the value 100. Later on we have assigned the value 10 to i in function abc(). This changes the value of parameter i of this function and not the value of the variable i. So WriteLine statement of this functon will print the value 100 and not 10.

Listing 11.24

```
//sharp611.cs
  class xyz68 {
   public static void Main()
{
   yyy a;
   int i=100;
   a=new yyy();
   a.alpha(out i);
   System.Console.WriteLine(i);
} }
   class yyy
{
   public void alpha(out int i)
{
   i=50;
}
 }
```

The output of this program will be displayed on the console as follows:


C:\>csc c:\sharp\sharp1\sharp61.cs
Microsoft (R) Visual C# Compiler Version
Copyright (C) Microsoft Corp 2000. All rights reserved.

This program solves the dilemma encountered in the previous example wherein we wanted the value given to the variable in function alpha to change as per statement and make this changed value available to function Main(). This has been made possible with the help of the variable scope modifier 'out'. By decaring a.alpha(out **i**) we are allowing function alpha to change the value of **i**. Similarly, the word **out** has been added in the function definition as well, so it knows **i** is going to change and hence **i** now changes in the original. Therefore WriteLine will display the new value to 10.

Listing 11.25

```
//sharp67.cs
   class xyz67 {
   public static void Main()
   {
    yyy a;
    int i=100;
    a = new yyy();
    a.alpha(i);
    System.Console.WriteLine(i);
   }
}
   class yyy
   {
     public void alpha(out int i)
     {
      i=10;
     }
}
```

This program is identical to the previous one except that it describes it if you forget to mention the word out in function with the following error message.

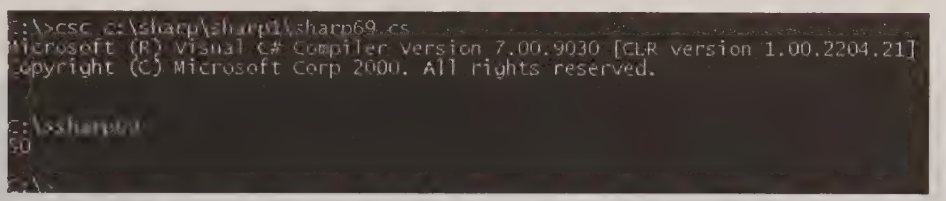
```
C:\>csc c:\sharp\sharp1\sharp67.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp67.cs(9,4): error CS1502: The best overloaded meth  match for
 'yyy.alpha(out int)' has some invalid arguments
sharp\sharp1\sharp67.cs(9,12): error CS1503: Argument '1': cannot convert from '
int' to 'out int'

C:\>
```

Listing 11.26

```
//sharp69.cs
   class hema {
   public static void Main()
   {
     yyy a;
```

```
        int i=100;
        a=new yyy();
        a.alpha(ref i);
        System.Console.WriteLine(i);
      }
}
      class yyy
      {
       public void alpha(ref int i)
       {
        i=50;
       }
}
```

The output of this program shall be displayed on the console as:


C:\>csc c:\sharp\sharp1\sharp69.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\sharp\>
50

This program uses the word ref instead of **out** used earlier. ref is identical to out in actual usage and we are bound to get a similar output. Here also the word ref must be used in two places, at the function call as a parameter and also at the actual function definition.

Listing 11.27

```
//sharp70.cs
   class xyz70 {
   public static void Main()
{
   yyy a;
   int i;
   a=new yyy();
   a.alpha(ref i);
   System.Console.WriteLine(i);
} }
   class yyy
{
   public void alpha(ref int j)
{
   j=50;
}
}
```

The output of this program on the console will be as follows:

```
C:\>csc c:\sharp\sharp1\sharp70.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp70.cs(9,16): error CS0165: Use of unassigned local variable 'i'

C:\>
```

The error message is due to the fact that the variable **i** is not initialized.

Listing 11.28

```
//sharp71.cs
  class xyz71 {
public static void Main()
{
   yyy a;
   int i;
   a=new yyy();
   a.alpha(out i);
   System.Console.WriteLine(i);
} }
   class yyy
{
   public void alpha(out int j)
{
   j=10;
} }
```

The output of this program will be displayed on the console as follows:

```
C:\>csc c:\sharp\sharp1\sharp71.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.


C:\>sharp71
10

C:\>
```

The program sharp71.cs is the same as sharp70.cs except that it uses **out** instead of **ref** . Here also, we have not initialised the variable i but there is no error message. This is the main difference between **out** and **ref**. In case of **ref** you must initialize the variable before changing its value.This is because you can read the value of the variable in case of **ref** and write to it .

Listing 11.29

```
//sharp72.cs
 class hema {
public static void Main()
{
```

```
        yyy a;
        int i=10;
        a=new yyy();
        a.alpha(out i);
        System.Console.WriteLine(i);
      }
}
      class yyy
      {
       public void alpha(out int j)
       {
         System.Console.WriteLine(j);
         j=10;
       }
}
```

The output of this program will be displayed on the console as follows:

```
C:\>csc c:\sharp\sharp1\sharp72.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

c:\sharp\sharp1\sharp72.cs(17,29): error CS0165: Use of unassigned local variable
'j'

C:\>
```

In this example, we have initialized the variable and used out, but still we get the error message. Here, we are first printing the value of `i` using the WriteLine statement which means that we are trying to read the value of `i` which is passed to the variable `j` in the function alpha. But with 'out', one cannot read values. This causes the error message.

11.19.1 Collection classes

This section shows how to implement a collection class that can be used with the **`foreach`** statement.

The **`foreach`** statement is a convenient way to iterate over the elements of an array.It can also enumerate the elements of a collection, provided that the collection class has implemented the **`System.Collections.IEnumerator`** and **`System.Collections.IEnumerable`** interfaces.

The following code sample illustrates writing a collection class to be used with foreach. The class is a string tokenizer, similar to the C run-time function strtok.

Listing 11.30

```
// Col.cs
using System;
using System.Collections;
public class Tokens : IEnumerable
{
```

```
    private string[] elements;
    Tokens(string source, char[] delimiters)
    {
          elements = source.Split(delimiters);
    }
    // IEnumerable Interface Implementation
    public IEnumerator GetEnumerator() {
    return new TokenEnumerator(this);
    }
    // Inner class implements IEnumerator interface
    private class TokenEnumerator : IEnumerator {
          private int position = -1;
          private Tokens t;
          public TokenEnumerator(Tokens t) {
                this.t = t;
          }
          public bool MoveNext() {
                if (position < t.elements.Length - 1) {
                      position++;
                      return true;
                 }
                else {
                      return false;
                }
          }
          public void Reset() {
           position = -1;
          }
          public object Current {
                get {
                      return t.elements[position];
                }
          }
    }
    // Test Tokens, TokenEnumerator
    static void Main() {
    Tokens f = new Tokens("Nicely prepared program.", new char[] {' ','-
'});
    foreach (string item in f) {
                Console.WriteLine(item);
          }
    }
}
```

The Output is as follows:

```
Nicely
prepared
program.
```

Note that Internally, tokens use an array, which implements IEnumerator and IEnumerable itself. The code sample could have leveraged the array's enumeration methods as its own, but that would have defeated the purpose of this example. In C#, it is not strictly necessary for a collection class to inherit from IEnumerable and IEnumerator in order to be compatible with foreach; as long as the class has the required GetEnumerator, MoveNext, Reset, and Current members, it will work with foreach. Omitting the interfaces has the advantage of allowing you to define the return type of Current to be more specific than object, thereby providing type-safety.

You can have the best of both—type-safety within C# and interoperability with other .NET languages—by inheriting from IEnumerable and IEnumerator and using explicit interface implementation, as demonstrated here.

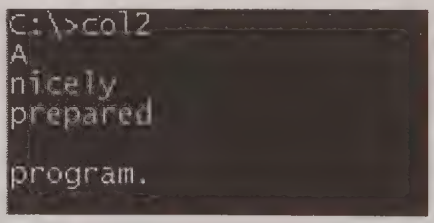
This sample is equivalent in function to the previously described code; but it provides additional type-safety in C# while maintaining interoperability with other languages.

Listing 11.31

```
// Col2.cs
using System;
using System.Collections;
public class Tokens: IEnumerable {
   private string[] elements;
   Tokens(string source, char[] delimiters) {
         elements = source.Split(delimiters);
   }
   // IEnumerable Interface Implementation
   public TokenEnumerator GetEnumerator()
   // non-IEnumerable
   version { return new TokenEnumerator(this); }
   IEnumerator IEnumerable.GetEnumerator() // IEnumerable
   version { return (IEnumerator) new TokenEnumerator(this); }
   // Inner class implements IEnumerator interface
   public class TokenEnumerator: IEnumerator { private int position = -
1; private Tokens t;
         public TokenEnumerator(Tokens t) { this.t = t; }
         public bool MoveNext() {
               if (position < t.elements.Length - 1) {
                     position++;      return true;
               }
               else {
                     return false;
               }
```

```
        }
        public void Reset() { position = -1; }
        public string Current // non-IEnumerator
        version: type-safe {
              get {
                    return      t.elements[position];
              }
        }
        object IEnumerator.Current
        // IEnumerator
        version: returns object {
              get {
                    return t.elements[position];
              }
        }
  }
  // Test Tokens, TokenEnumerator
  static void Main() {
        Tokens f = new Tokens("A nicely prepared  program.", new char
[]  {'                    ',','-'});
        foreach (string item in f) // try changing string to int
{                       Console.WriteLine(item);
        }
  }
}
```

The output of this program shall get displayed as follows:



11.20 foreach Statement

The foreach statement repeats a group of embedded statements for each element in an array or an object collection. The foreach statement is used to iterate through the collection to get the desired information, but should not be used to change the contents of the collection to avoid unpredictable side effects. The statement assumes the following form:

foreach (type identifier in expression) statement where type is the type of identifier.

identifier: The iteration variable that represents the collection element.

expression:Object collection or array expression. The type of the collection element must be convertible to the identifier type.

statement: The embedded statement(s) to be executed.

Remarks: The embedded statements continue to execute for each element in the array or collection. After the iteration has been completed for all the elements in the collection, control is transferred to the statement immediately following the foreach block.

When used with an array, the foreach statement repeats the embedded statement(s) for each element in the array.

In this example, an array of integers is searched for even and odd numbers. A counter for each type of number stores the number of occurrences.

Listing 11.32

```
// Using foreach with arrays
col3.cs
using System;
  class MainClass {
        public static void Main()
        {
              int odd = 0, even = 0;
              int[] arr = new int [] {2,9,44,1,7,8,13,25};
              foreach (int i in arr)
                    {
                          if (i%2 == 0)
                                even++;
                          else
                                odd++;
                    }
                    Console.WriteLine("Found {0} Odd Numbers, and {1}
Even  Numbers.",
                       odd, even) ;
              }
    }
```

The output of this program will be:

```
C:\>col3
Found 5 Odd Numbers, and 3 Even Numbers.
```

11.20.1 Using foreach with Collections

To iterate through a collection, the collection must meet specific requirements. For example, in the following foreach statement:

Foreach (ItemType item in myCollection) myCollection must meet the following requirements:

1. The collection type:
2. Must be one of the types: interface, class, or struct.
3. Must include an instance method named GetEnumerator that returns a type, for example, Enumerator (explained below).
4. The type Enumerator (a class or struct) must contain:
5. A property named Current that returns ItemType or a type that can be converted to it. The property accessor returns the current element of the collection.

6. A bool method, named MoveNext, that increments the item counter and returns true if there are more items in the collection.

There are three approaches to using collections:

1. Create a collection using the above instructions. This collection can only be used only in a C# program.
2. Create a generic collection using the above instructions, in addition to implementing the IEnumerable interface. This collection can be used from other languages such as Visual Basic.
3. Use one of the predefined collections in the collection classes.

The ensuing examples demonstrate these three approaches.

Example 1:
This example creates a collection by using the instructions above.

Example 2:
This example creates a collection by using the instructions above and implements the IEnumerable and IEnumerator interfaces.

Example 3:
Using one of the predefined collection classes. This example creates a Hashtable instance and uses the Hashtable class members to manipulate the collection. The Hashtable class represents a dictionary of associated keys and values, implemented as a hash table.

Listing 11.33

```
// Col4.cs
using System;
// Declare the collection: public class MyCollection {
        int[] items;
        public MyCollection()
        {
              items = new int[5] {125, 440, 380, 342, 750};
        }
        public MyEnumerator GetEnumerator()
        {
              return new MyEnumerator(this);
        }
        // Declare the enumerator class:
        public class MyEnumerator
        {
              int nIndex;
              MyCollection collection;
              public MyEnumerator(MyCollection coll)
              {
                    collection = coll;
                    nIndex     = -1;
```

```
            }
            public bool MoveNext()
            {
                  nIndex++;
                  return(nIndex < collection.items.GetLength(0));
            }
            public int Current
            {
                  get
                  {
                        return(collection.items[nIndex]);
                  }
            }
      }
}
public class MainClass {
      public static void Main()
{
      MyCollection col = new MyCollection();
      Console.WriteLine("Values shown  in this collection are:");
      // Display collection items:
      foreach (int i in col)
      {
            Console.WriteLine(i);
      }
}
}
```

The output of this program will be:

```
C:\>col4
Values shown  in this collection are:
125
440
380
342
750
```

Example:
This example repeats the same algorithm of Example 1, but uses a generic collection that can be enumerated from other languages such as Visual Basic. This type of collection must implement the IEnumerable interface from the System.Collections namespace.

Listing 11.34

```
// Col5.cs
using System;
using System.Collections;
// Declare the collection and implement the IEnumerable  interface:
public class MyCollection: IEnumerable {
        int[ ] items;
        public MyCollection()
        {
              items = new int[5] {100, 450, 380, 75, 574};
        }
        public MyEnumerator GetEnumerator()
        {
              return new MyEnumerator(this);
        }
        // Implement the GetEnumerator() method:
        IEnumerator IEnumerable.GetEnumerator()
        {
              return GetEnumerator();
        }
    // Declare the enumerator and implement the IEnumerator interface:
        public class MyEnumerator: IEnumerator
        {
              int nIndex;
              MyCollection collection;
              public MyEnumerator(MyCollection coll)
              {
                    collection = coll;
                    nIndex = -1;
              }
              public void Reset()
              {
                    nIndex = -1;
              }
              public bool MoveNext() {
                    nIndex++;
                    return(nIndex < collection.items.GetLength(0));
              }
              public int Current
              {
                    get
                    {
                          return(collection.items[nIndex]);
                    }
              }
```

```
                // The current property on the IEnumerator interface:
                object IEnumerator.Current
                {
                      get
                      {
                            return(Current);
                      }
                }
          }
    }
    public class MainClass {
    public static void Main(string [] args)
    {
       MyCollection col = new MyCollection();
       Console.WriteLine("Amount shown in this collection are:");
       // Display collection items:
       foreach (int i in col)
       {
          Console.WriteLine(i);
       }
    }
}
```

The output of this program will be:

```
C:\>col5
Amount shown in this collection are:
100
450
380
75
574
```

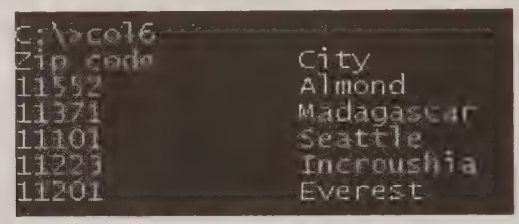
In this example, the predefined Hashtable collection class is used. By using the System.Collections namespace in your program, you can have access to the Hashtable class and its members. To add entries to the Hashtable object, use the Add method.

Listing 11.35

```
//Col6.cs    Using the Hashtable collection class
using System;
using System.Collections;
public class MainClass {
   public static void Main(string [] args)
   {
      // Declare a Hashtable object:
      Hashtable ziphash = new Hashtable();
      // Add entries using the Add() method:
      ziphash.Add("11223", "Incroushia");
```

```
        ziphash.Add("11552", "Almond");
        ziphash.Add("11201", "Everest");
        ziphash.Add("11101", "Seattle");
        ziphash.Add("11371", "Madagascar");
        // Display contents:
        Console.WriteLine("Zip code\tCity");
        foreach (string zip in ziphash.Keys)
        {
            Console.WriteLine(zip + "\t\t" + ziphash[zip]);
        }
    }
}
```

The output of this program will be



11.20.2 Features of the foreach Statement

As mentioned earlier, the foreach statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

The type and the identifier of a foreach statement declare the iteration variable of the statement. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a foreach statement, the iteration variable represents the collection element for which an iteration is currently being sought. A compile-time error occurs if the embedded statement attempts to assign to the iteration variable or pass the iteration variable as a ref or out parameter.

The type of the expression of a foreach statement must be a collection type (as defined below), and an explicit conversion must exist from the element type of the collection to the type of the iteration variable.

C contains a public instance method with the signature GetEnumerator() that returns a struct-type, class-type, or interface-type, in the following called E.

A type C is said to be a collection type if all the following conditions are satisfied.

1. E contains a public instance method with the signature MoveNext() and the return type bool.
2. E contains a public instance property named Current that permits reading. The type of this property is said to be the element type of the collection type. The System.Array type is a collection type, and since all array types derive from System.Array, any array type expression

is permitted in a foreach statement. For single-dimensional arrays, the foreach statement enumerates the array elements in increasing index order, starting with index 0 and ending with index Length -1. For multi-dimensional arrays, the indices of the rightmost dimension are increased first.

A foreach statement is executed as follows:

1. The collection expression is evaluated to produce an instance of the collection type. This instance is referred to as c in the following. If c is of a reference-type and has the value null, a NullReferenceException is thrown.
2. An enumerator instance is obtained by evaluating the method invocation c.GetEnumerator(). The returned enumerator is stored.

In a temporary local variable, in the following referred to as e, it is not possible for the embedded statement to access this temporary variable. If e is of a reference-type and has the value null, a NullReferenceException is thrown.

3. The enumerator is advanced to the next element by evaluating the method invocation e.MoveNext().
4. If the value returned by e.MoveNext() is true, the following steps are performed:
5. The current enumerator value is obtained by evaluating the property access e.Current, and the value is converted to the type of the iteration variable by an explicit conversion. The resulting value is stored in the iteration variable such that it can be accessed in the embedded statement.
6. Control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement.

(possibly from execution of a continue statement), another foreach iteration is performed, starting with the step above that advances the enumerator. If the value returned by e.MoveNext() is false, control is transferred to the end point of the foreach statement.

Within the embedded statement of a foreach statement, a break statement may be used to transfer control to the end point of the foreach statement (thus ending iteration of the embedded statement), and a continue statement) may be used to transfer control to the end point of the embedded statement (thus executing another iteration of the foreach statement).

The embedded statement of a foreach statement is reachable if the foreach statement is reachable. Likewise, the end point of a foreach statement is reachable if the foreach statement is reachable.

The following is a complete C# program that declares and instantiates arrays as discussed.

Listing 11.36

```
// Col9.cs
 using System;
 class DeclareArraysSample
 {
     public static void Main()
    {
          // Single-dimensional array
```

```
            int[ ] numbers = new int[5];

            // Multidimensional array
           string[,]  names = new string[5,4];

            // Array-of-arrays (jagged array)
            byte[ ][ ] scores = new byte[5][ ];

            // Create the jagged array
            for (int i = 0; i < scores.Length; i++)
            {
                scores[i] = new byte[i+3];
            }

            // Print height of each row
          for (int i = 0; i < scores.Length; i++)
          {
             Console.WriteLine("Height of the row no. {0} is {1}", i,
scores[i].Length);
          }
      }
}
```

The output of this program will be displayed as follows:

```
C:\>col9
Height of the row no. 0 is 3
Height of the row no. 1 is 4
Height of the row no. 2 is 5
Height of the row no. 3 is 6
Height of the row no. 4 is 7
```

12

STRINGS

12.1 Strings

All *strings* in C# are instances of the **System.String** type in the Common Language Runtime. Because of this, many built-in operations that work with strings are available. For example, the String class defines an indexer function that can be used to iterate over the characters of the string.

Listing 12.1

```
//string1.cs
using System;
class Test {
  public static void Main()
  { string s = "Testing String";
     for (int index = 0; index < s.Length;
   index++)

   Console.WriteLine("char:{0}", s[index]);
  }
}
```

The output of this program shall get displayed as:

```
C:\>csc c:\temp\vkj\string.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

temp\vkj\string.cs(6,34): error CS0117: 'string' does not contain a definition f
or 'length'
temp\vkj\string.cs(8,4): error CS0234: The type or namespace name 'console' does
 not exist in the class or namespace ''
```

12.2 String Operations

The string class is an example of an immutable type, which means that the characters contained in the string cannot be modified by users of the string. All operations performed by the string class return a modified version of the string rather than a modified instance on which the method is called.

The String class supports the following comparison and searching methods:

Item	DESCRIPTION
Compare()	Compares two strings
CompareOrdinal()	Compares two string regions using an ordinal comparison
CompareTo()	Compares the current instance with another instance
EndsWith()	Determines whether a substring exists at the end of a string
StartsWith()	Determines whether a substring exists at the beginning of a string
IndexOf()	Returns the position of the first occurrence of a substring
LastIndexOf()	Returns the position of the last occurence of a substring

The String class supports the following modification methods:

ITEM	DESCRIPTION
Concat()	Concatenates two or more strings or objects together. If objects are passed, the ToString() function is called on them.
CopyTo()	Copies a specified number of characters from a location in this string into an array.
Insert()	Returns a new string with a substring inserted at a specific location
Join()	Joins an array of strings together with a separator between each array element.
PadLeft()	Left aligns a string in field
PadRight();	Right aligns a string in a field.
Remove()	Deletes characters from a string
Replace();	Replaces all instances of a character with a different character
Split();	Creates an array of strings by splitting a string at any occurrence of one or more characters
Substrng()	Extracts a substring from a string
ToLower();	Returns a lower-case version of a string
ToUpper();	Returns an upper-case version of a string
Trim();	Removes white space from a string
TrimEnd();	Removes a string of characters from the end of a string
TrimStart();	Removes a string of characters from the beginning of a string.

12.3 Converting Objects to Strings

The function **object.ToString()** is overridden by the built-in types to provide an easy way of converting from a value to a string representation of that value. Calling **ToString()** produces the default representation of a value; a different representation may be obtained by calling **String.Format()**.

The split function can be used to break a string into substrings at separators.

Listing 12.2

```
//objects1.cs
using System;
class Test {
   public static void Main()
   {
     string s = "C#, is better than other oops";
     char[] separators = new char[] {' ', ' ',};
     foreach (string sub in s.Split(separators))
   {
     Console.WriteLine("Word: {0}", sub);
    }
  }
}
```

The output of this program then is as follows:

```
C:\>objects1
Word: C#,
Word: is
Word: better
Word: than
Word: other
Word: oops
```

The separators character array defines what characters the string will be broken on. The Split() function returns an array of strings, and the foreach statement iterates over the array and prints it out.

In this case, the output isn't particularly useful because the "," string gets broken twice. This can be fixed by using the regular expression classes.

12.4 StringBuilder

Though the `String.Format()` function can be used to create a string based on the values of other strings, the StringBuilder class provided by the run-time makes this process easier. The StringBuilder class supports the following properties and methods:

PROPERTY	DESCRIPTION
Capacity	Retrieves or sets the number of characters the StringBuilder can hold
[]	The StringBuilder indexer is used to get or set a character at a specific position.
Length	Retrieves or sets the length
MaxCapacity	Retrieves the maximum capacity of the StringBuilder

METHOD	DESCRIPTION
Append()	Appends the string representation of an object.
AppendFormat()	Appends a string representation of an object, using a specific format string for the object.
EnsureCapacity()	Ensures the StringBuilder has enough room for a specific number of characters
Insert()	Inserts the string representation of a specified object at a specified position
Remove()	Removes the specified characters
Replace()	Replace all instances of a character with a new character.

The following example shows how the StringBuilder class can be used to create a string from separate strings.
Listing 12.3

```
//objects2.cs
using System;
using System.Text;
class Test {
   public static void Main()
{
   string s = "C# have all good qualities of java and c++";
   char[] separators = new char[] {' ', ' ',};
   StringBuilder sb = new StringBuilder();
   int number = 1;
   foreach (string sub in s.Split(separators))
{
   sb.AppendFormat("{0}: {1}", number++, sub);
}
   Console.WriteLine("{0}", sb);
}
}
```

The output of this program thus, gets displayed on the console as:

```
C:\>objects2
1: C#2: have3: all4: good5: qualities6: of7: java8: and9: c++
```

12.5 Regular Expressions

If the searching functions found in the String class aren't powerful enough, the System.Text namespace contains a regular expression class named Regex. Regular expressions provide a very powerful method for doing search and / or replace functions.

While there are a few examples of using regular expression in this section, a detailed description of them is beyond the scope of the book. There are several regular expression books available, and the subject is also covered in most books on Perl.

The regular expression class uses a unique technique to get maximum performance. Rather than interpret the regular expression for each match, it writes a short program on the fly to implement the regular expression match, and that code is then run.

The previous example(Listing 12.3) using Split() can be revised to use a regular expression, rather than single characters, to specify how the split should occur. In the regular expression, the string is split either on a space or on a comma followed by a space.

The FileStream class is derived from Stream class. This class can be used for reading from and writing to files such as bytes, characters, strings, and other data-type values to a file. Other class used in this sample is StreamWriter. The StreamWriter class's Write and WriteLine members write to a file.

In this sample code, FileStream class is used to create a text file and StreamWrite to write text to the text file.

Listing 12.4

```
namespace mcFileStream
{
   using System;
   using System.IO;
   public class mcLogFile
   {
         public static void Main(String[] args)
         {
               // Create a text file C:\temp\mcb.txt
               FileStream fs = new FileStream(@"c:\temp\mcb.txt" ,
                     FileMode.OpenOrCreate, FileAccess.Write);
               StreamWriter m_streamWriter = new StreamWriter(fs); //
Write to                                                    //the  file
using
               StreamWriter class m_streamWriter.BaseStream.Seek(0,
                           SeekOrigin.End);
               m_streamWriter.Write(" File Write Operation Starts :
");
               m_streamWriter.WriteLine("{0} {1}",
               DateTime.Now.ToLongTimeString(),
                           DateTime.Now.ToLongDateString());
               m_streamWriter.WriteLine(" First Line : Data is first
line \n");
               m_streamWriter.WriteLine(" This is next line in the
text file.                                  \n  ");
               m_streamWriter.Flush(); // Read from the file using
StreamReader                          //class
```

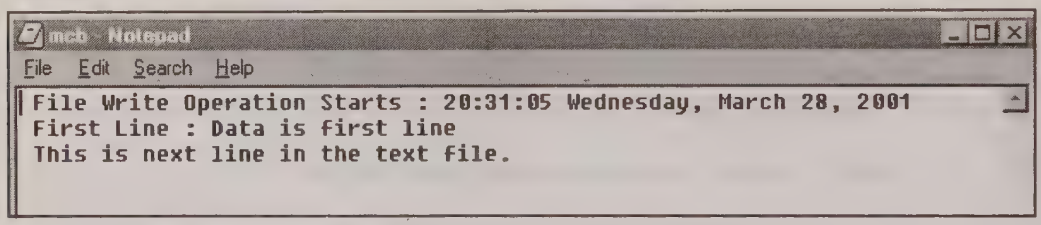
```
                //
                StreamReader m_streamReader = new StreamReader(fs); //
                string str = m_streamReader.ReadLine();
            }
        }
    }
```

when compiled, runs as shown below:

```
C:\>csc \file.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>file
```

The above program will create a file by name C:\temp\mcb.txt in your computer if directory temp already exist.

The contents of this file is as follows:


mcb - Notepad
File Edit Search Help
File Write Operation Starts : 20:31:05 Wednesday, March 28, 2001
First Line : Data is first line
This is next line in the text file.

12.6 File and Folder Operations

Most often, we need to access the file system, read and write to files, move or copy files around, or explore folders to check what files are there. The .NET base classes include a number of classes that provide a rich set of functionality to do these tasks. These classes are contained in the System.IO namespace. Since the AppWizard doesn't add code to refer to this namespace by default, we add the line using System.IO near the top of the Form1.cs source file for all the samples in this section:

```
namespace prog.SampleCode.CSharpPreview.ChBaseClasses {
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.WinForms;
using System.Data;
using System.IO;
```

Note that (as you can find out from the WinCV tool) these classes are still defined in mscorlib.dll, so we don't need to add any files to the project references.

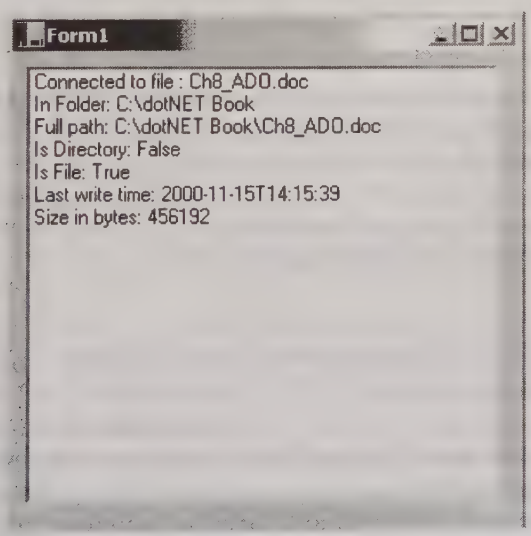
12.7 Finding Out Information About a File

Our first file operations sample, which we will call FileInfo will demonstrate how to get information about a file, such as its last write time, and its size. To do this, we use a class, System.IO.File. The code to obtain the information is as follows:

```
File fl = new File("C:\\dotNET Book\\Ch8_ADO.doc");
AddItem("Connected to file : " + fl.Name);
AddItem("In Folder: " + fl.Directory);
AddItem("Full path: " + fl.FullName);
AddItem("Is Directory: " + fl.IsDirectory.ToString());
AddItem("Is File: " + fl.IsFile);
AddItem("Last write time: " + fl.LastWriteTime.ToString()); AddItem("Size
in bytes: " + fl.Length);
```

This code should be fairly self-explanatory. We construct a File **instance** by supplying the full path name of the location of the file in the file system, and the File instance then refers to that file, and we can simply read off a number of properties of it. In this case, we are binding to the file for one of the other chapters of this book, the ADO+ chapter. Note that \ by itself is interpreted as the start of an escape sequence in strings in C#, so we need to use \\ to represent a single backslash in the pathname. You can also use an alternative syntax, in which the @ character precedes the string, which indicates that characters should not be escaped. Hence you would write:

```
File fl = new File(@"F:\dotNET Book\Ch8_ADO.doc")
```



In general, you can use a File object to connect to either a file or a folder. But if you connect to a folder and then attempt to access those properties, the folder (such as Length or LastWriteTime) will raise an exception.

12.8 Listing Files in a Folder

To explore the contents of a folder, we need another base class—in this case the class Directory is also in the System.IO namespace. Note that the .NET base classes generally refer to folders as directories in class and method names. This corresponds to normal terminology on web sites and on Unix and Linux machines, as well as on Windows 3.1 when it was around, but can be confusing if you are used to Windows terminology, in which a folder is an item in the file system that contains files, and a directory is a more sophisticated complete information store (such as Active Directory). We will continue to use the term folder in this chapter, in accordance with normal usage for the Windows file system.

The following code sample connects to the folder F:\dotNET Book and separately lists the files and folders in it.

```
Directory fl = new Directory("C:\\dotNET Book");
AddItem("Connected to folder: " + fl.Name);
AddItem("Full path: " + fl.FullName);
AddItem("Is Directory: " + fl.IsDirectory.ToString());
AddItem("Is File: " + fl.IsFile);
AddItem(""); AddItem("Files contained in this folder:");
File [] childfiles = fl.GetFiles();
foreach (File childfile in childfiles) { AddItem(" " + childfile.Name);
}
AddItem(""); AddItem("Subfolders contained in this folder:");
Directory [] childfolders = fl.GetDirectories();
foreach (Directory childfolder in childfolders) { AddItem(" " +
childfolder.Name);
}
```

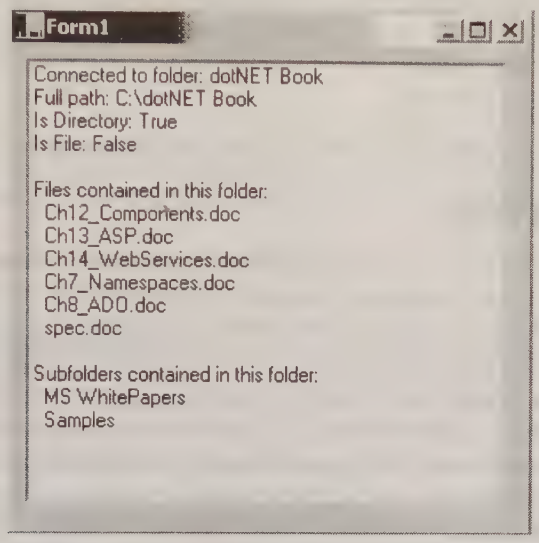
This code starts off by showing that the way of connecting to a folder is the same as for a file, and that the Directory and File classes both share the Boolean IsDirectory and IsFile properties, which can be used to distinguish what the object is if you are unsure. This means that if you do not know what an object is, you can, for example, bind to it as a file, then use the IsDirectory property to check if it is actually a folder—and re-bind to it as a Directory if IsDirectory returns true. The resulting code would look something like this:

Listing 12.5

```
File fl = new File("C:\\DotNET Book");
if (fl.IsDirectory == true) { fl = null; Directory dr = new
Directory("C:\\DotNET Book");
// process as directory } else { // process as file }
```

In the above code, we next use a method, GetFiles() to retrieve a list of the files in the directory. This method returns an array of File instances, each one already bound to a file—so we can use a foreach loop to iterate through the array and carry out whatever processing we need to do on each file. The Directory class has another method, GetDirectories(), which works in exactly the same way as GetFiles(), but returns an array of Directory instances that refer to each subfolder. In both cases in the sample, we use these methods to simply display the names of the files and folders.

This code produces the following output on the console:



12.9 Copying and Deleting files

In the next sample, we really will get to know the directory system. It is a Windows project in which we add the code to the form's constructor, though in this case there is no real output to display.

We start off by binding to the dotNETBook folder, and we create both a new empty file and a new empty subfolder there.

```
Directory fl = new Directory("C:\\dotNET Book");
fl.CreateSubdirectory("SampleBackups");
fl.CreateFile("MyNewFile.txt");
```

We are not going to write anything to the file yet—we will do that later on.
Next we bind to one of the files in the f:\dotNET Book folder, rename it and copy it:

```
File adofile = new File("C:\\dotNET Book\\Ch8_ADO.doc");
adofile.CopyTo("C:\\dotNET Book\\SampleBackups\\Ch8_Backup.doc");
```

Note that you should put the complete path in, otherwise the file will be copied to the same directory as the executable.

Now we have a go at deleting things—first the file spec.doc and then the Samples folder.

```
File sparefile = new File("C:\\dotNET Book\\spec.doc");
sparefile.Delete();
Directory sparefolder = new Directory("C:\\dotNET Book\\Samples");
sparefolder.Delete(true);
```

The **File.Delete()** method does not take any parameters. The **Directory.Delete()** method has two overloads. One (which we haven"t demonstrated here) takes no parameters and does a simple delete and the file or directory goes to the recycle bin. The other takes one parameter—a Boolean which indicates whether the delete operation is recursive. In this case, we have specified that it is. If we had wanted the delete not to be recursive, we would have written:

```
sparefolder.Delete(false);
```

In that case, if sparefolder contained subfolders an exception would be raised.

12.10 Reading Text Files

Reading files is quite simple, since Microsoft have provided a large number of classes that represent streams, which may be used to transfer data. Transferring data here can include such things as reading and writing to files, or downloading from the Internet, or simply moving data from one location to another using a stream.

The available classes are all derived from the class **`System.IO.Stream`**, which can represent any stream, and the various classes represent different specializations, for example streams that are specifically geared to reading or writing to files. In general, for the examples in this chapter that involve using streams, there are potentially a number of different ways to write the code, using any of several of the available stream objects. However, in this chapter, we will just present one way that you could perform each of the processes of reading and writing to text and binary files.

Reading text files is quite simple—for this we are going to use the StreamReader class. The StreamReader represents a stream specifically geared to reading text. We will demonstrate the process with a sample that reads in and displays the contents of the ReadMe.txt file generated by the developer environment's AppWizard for our earlier EnumFiles sample. The code appears as:

Listing 12.6

```
File fIn = new File("C:\\dotNET
Projects\\Namespaces\\EnumFiles\\ReadMe.txt");
StreamReader strm = fIn.OpenText();
// continue reading until end of file sring sLine;
do { sLine = strm.ReadLine();
AddItem(sLine);
} while (sLine != null);
strm.Close();
```

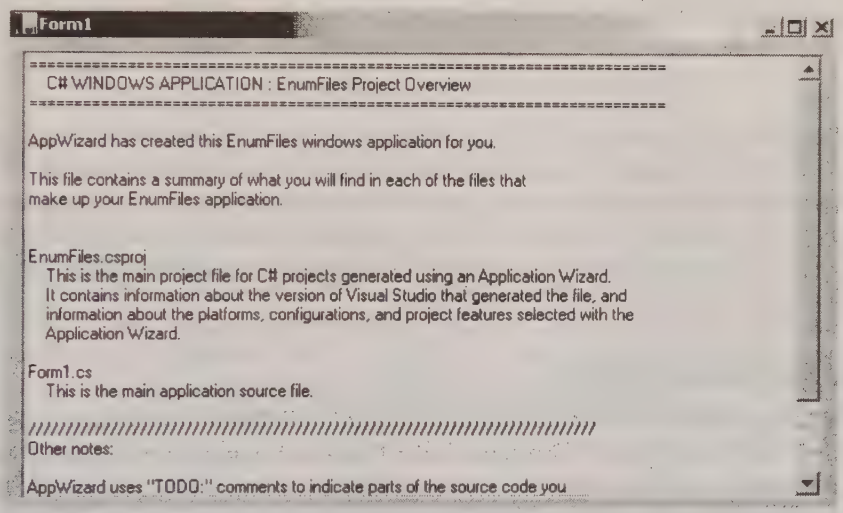
We obtain a StreamReader instance using the OpenText() method of the File class. The StreamReader class contains several methods that either read or peek at differing amounts of data.

Peeking means looking ahead at the data, but without actually moving through the data. The best way to understand this is by imagining a pointer that indicates which bit of the file you are due to read next. If you read data, then the pointer will be moved to point at the byte following the last byte read, so the next read (or peek) will bring in the next block of data. If you peek at data, the pointer is not changed, so the next read (or peek) will retrieve the same data again.

The most useful method however for our purposes is ReadLine(), which reads as far as the next carriage return, returning the result in a string. If we have reached the end of the file, ReadLine() does not throw an exception, but simply returns a null reference—so we use this to test for the end of the file. Note that a null reference is not the same as an empty string. If we had instead applied the condition:

```
while (sLine != "");
```

to the do loop, the loop would have finished the moment we came to a blank line in the file, not at the end of the file. (StreamReader.ReadLine() returns the string without the trailing carriage return and line feed).
Running this sample produces the following output:


Form1
C# WINDOWS APPLICATION : EnumFiles Project Overview
AppWizard has created this EnumFiles windows application for you.
This file contains a summary of what you will find in each of the files that make up your EnumFiles application.
EnumFiles.csproj
This is the main project file for C# projects generated using an Application Wizard. It contains information about the version of Visual Studio that generated the file, and information about the platforms, configurations, and project features selected with the Application Wizard.
Form1.cs
This is the main application source file.
Other notes:
AppWizard uses "TODO:" comments to indicate parts of the source code you

12.11 Writing Text Files

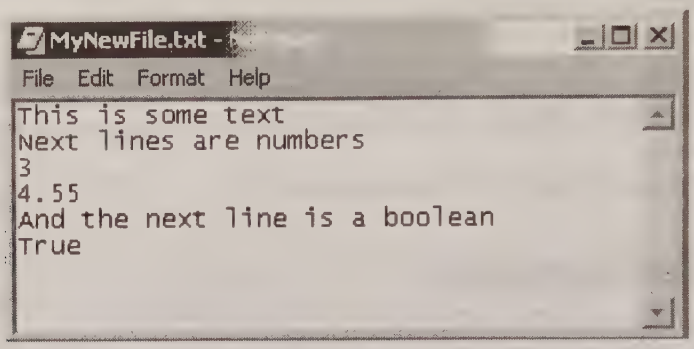
Writing text files follows similar principles to reading from them—in this case, we use the StreamWriter class.

What makes writing text files even easier than reading them is that the method we use to write a line of text output followed by a carriage return-line feed, StreamWriter.WriteLine(), has a number of overloads so we don't necessarily need to pass it just text. It will accept a string, an object, a Boolean or several of the numeric types. This can be seen from this code sample, which writes out some text to the blank file we created earlier, MyNewFile.txt.

```
StreamWriter strm = new StreamWriter("C:\\dotNET Book\\MyNewFile.txt",
false);
strm.WriteLine("This is some text");
strm.WriteLine("Next lines are numbers");
strm.WriteLine(3);
strm.WriteLine(4.55);
```

```
strm.WriteLine("And the next line is a boolean");
strm.WriteLine(true); strm.Close();
```

The results of this can be seen in Notepad:



There are a number of overrides to the constructor of the StreamWriter. The constructor we have picked in our code sample is quite flexible, taking two parameters: the full name of the file, and a Boolean that indicates whether data should be appended to the file. If this is false, then the contents of the file will be overwritten by the StreamWriter. In either case, the file will be opened if it already exists or created if it doesn't. This behavior can be customized by using other more complex constructors to the StreamWriter.

12.12 Reading Binary Files

Once we get to binary files we need to abandon the text-specific StreamReader and StreamWriter classes in place of a more general- purpose class. There are actually two classes that will do the job, System.IO.Stream and System.IO.FileStream. FileStream is designed specifically for reading and writing to files, while Stream is able to transmit data between files or other objects. The two classes work in very similar ways, and for the sake of demonstrating both of them, we'll use Stream for reading data and then use FileStream in the following sample which writes data to a file.

This sample demonstrates how to use the Stream class to read data. It opens a file and reads it—a byte at a time—each time displaying the numeric value of the byte read.

```
File fl = new File("C:\\dotNET Book\\TestReader.txt");
Stream strm = fl.OpenRead();
int iNext;
do { iNext = strm.ReadByte();
if (iNext != -1) AddItem(iNext.ToString());
} while (iNext != -1);
strm.Close();
```

We obtain an instance of the Stream class by first opening instantiating a File object attached to the required file and calling its **OpenRead()** method. We then read through the file by calling the **Stream.ReadByte()** method. This method reads the next byte returning its value as an

int. If we have reached the end of the file, then -1 is returned, but no exception is thrown—and it is this condition we use to test for the end of the file. Note that the Stream class also has a **Read()** method which can read a specified number of bytes in one go—we have chosen to use **ReadByte()** here as it leads to simpler code.

The file we have tested this code on looks like this:the first few letters of the alphabet followed by three carriage return-line feeds. Although the code will work on any file, a text file was chosen because that makes it easier to see visually that the results are correct.

Running the code on this file produces the following output:

12.13 Writing Binary Files

Although we can use either the Stream or FileStream classes to perform this task, we will use an instance of FileStream for this sample. This code writes out a short text file that contains the letters FGHIJK followed by a carriage return-line feed combination. Note here that although this code is capable of writing any data, we are using textual data in the sample so that we can easily use Notepad to check that the sample has worked.

The code will be as follows:

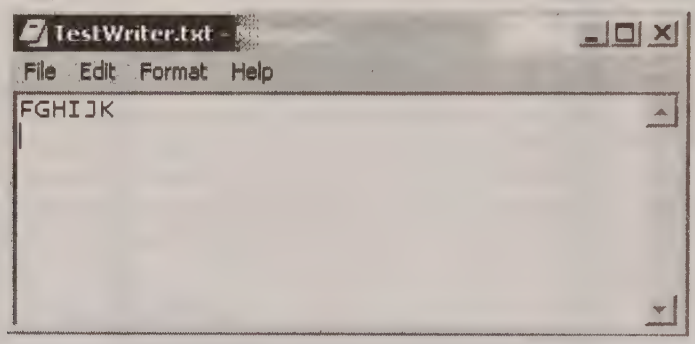
```
byte [] bytes = {70,71,72,73,74,75,13,10};
FileStream strm = new FileStream("C:\\dotNET  Book\\TestWriter.txt",
FileMode.OpenOrCreate, FileAccess.Write);
foreach (byte bNext in bytes) { strm.WriteByte(bNext);
} strm.Close();
```

We first define an array of bytes that contains the data to be written to the file—in this case, the ASCII codes of the characters.True binary data would have simply meant changing some of the values in this array to represent non-printable characters.

Next, we instantiate a FileStream object. The constructor we use takes three parameters: the full pathname of the file, the mode we are using to open it and the access required. The mode and access merit more consideration—they are enumerated values respectively taken from two further classes in the System.IO namespace: FileMode and FileAccess. The possible values these can take are all self-explanatory. In the case of the mode the possible values are Append, Create, CreateNew, Open, OpenOrCreate and Truncate. For the access they are Read, ReadWrite and Write.

Finally, we use the WriteByte method of the FileStream object to write out each byte before closing the file. Again there is a FileStream.Write method, which can write out a number of bytes at a time and which you may prefer to use. We have stuck with WriteByte because it makes it clearer what is going on as we loop through the array.

And finally, we test whether this code has worked: After running it, opening the new file in Notepad gives this:



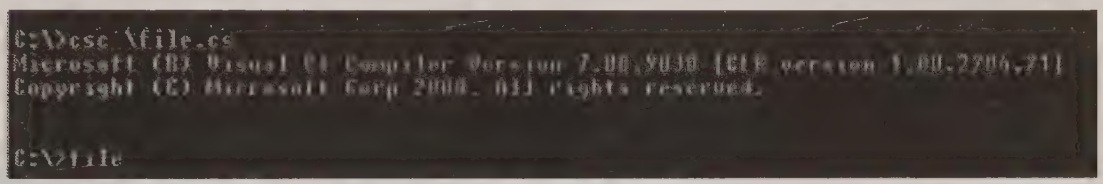
12.13.1 Creating A File

Listing 12.7

```
namespace mcFileStream {
using System;
using System.IO;
public class mcLogFile
   {
         public static void Main(String[] args)
         {
               // Create a text file C:\temp\mcb.txt
               FileStream fs = new FileStream(@"c:\temp\mcb.txt" ,
               FileMode.OpenOrCreate, FileAccess.Write);
               StreamWriter m_streamWriter = new StreamWriter(fs);
               //    Write to the file using StreamWriter class
               m_streamWriter.BaseStream.Seek(0, SeekOrigin.End);
               m_streamWriter.Write(" File Write Operation Starts :
                                    ");
```

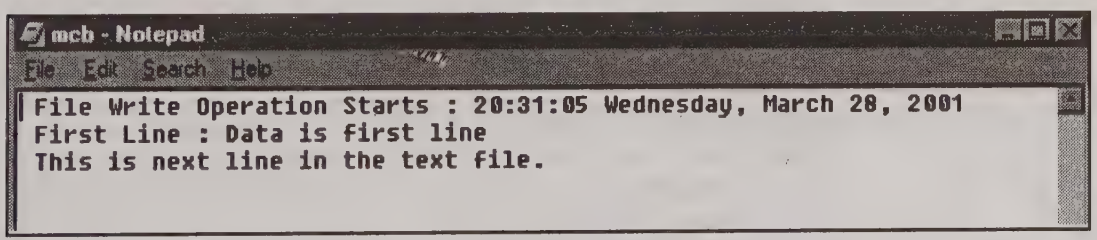
```
                m_streamWriter.WriteLine("{0} {1}",
                DateTime.Now.ToLongTimeString(),
                      DateTime.Now.ToLongDateString());
                m_streamWriter.WriteLine(" First Line : Data is first
                                  line \n");
                m_streamWriter.WriteLine(" This is next line in the
                      text file. \n  ");
                m_streamWriter.Flush();
                // Read from the file using StreamReader class
                //
                StreamReader m_streamReader = new StreamReader(fs);
                //
                string str = m_streamReader.ReadLine();
            }
        }
    }
```

when compiles and runs is as follows:


C:\>csc \file.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>file

The above program will create a file by name C:\temp\mcb.txt in your computer if directory temp already exist.

The contents of this file is as follows:


mcb - Notepad
File Edit Search Help
File Write Operation Starts : 20:31:05 Wednesday, March 28, 2001
First Line : Data is first line
This is next line in the text file.

12.13.2 Strings

String is the type of strings, including literals, in the .NET Framework. String objects are immutable, meaning once they're created their values cannot be changed. (However, you can reassign the string reference to another string, freeing up the first string for garbage collection if no other references to it exist.)

The methods of String that appear to manipulate the string do not change the current string; instead, they create a new string and return it. Note that the process of repeatedly creating and throwing away strings can be slow. But making strings immutable has a number of advantages in that ownership, aliasing, and threading issues are all much simpler with immutable objects.

Enters StringBuilder. A StringBuilder object is not a string, but is used to manipulate characters. It contains a buffer, typically initialized with a string but usually larger than that string. This buffer can be manipulated in place without creating a new string—you can insert, append, remove, and replace characters. When you're done manipulating the characters, use StringBuilder's ToString method to extract the finished string from it.

Both these types contain Unicode characters of type Char and support an indexer that returns Char. Because String is immutable, the indexer is read-only. Of course, the StringBuilder indexer is read/write.

String String is a sealed class, meaning you cannot inherit from it (and also meaning the system can do certain optimizations to make string processing faster).

Most .NET languages will have some built-in support for strings. For instance, they typically support string literals and operations such as concatenation. But the syntax for that will vary from language to language.

12.13.3 String literals and operations in C#

C# supports two types of string literals: 'regular' string literals and 'verbatim' string literals.

Regular string literals are similar to string literals in C and C++: They are delimited by quotation marks and can contain escape sequences to represent various control characters and any Unicode character. They may not extend to a new line, but adjacent string literals (separated only by white space) are concatenated. The escape sequences all begin with a backslash ("\") and include:

```
\t              tab (Unicode 0x0009)
\r              carriage return (0x000d)
\n              newline (line feed) (0x000a)
\v              vertical tab (0x000b)
\a              alert (0x0007)
\b              backspace (0x0008)
\f              form feed (0x000c)
\0              null (0x0000)
\\              backslash (0x005c)
\'              single quote (0x0027)
\"              double quote (0x0022)
\u              ABCD Unicode character 0xABCD
(where A, B, C, and D are valid hexadecimal digits 0-9, a-f, A-F)
```

All of these escape sequences can be used in any regular string literal or in any character literal (such as '\t'). In addition, the Unicode escape sequences can be used in identifiers, so 'if (a\u0066b == true'is the same as 'if (afb == true)' (since Unicode 0x0066 is 'f'). This allows you to write identifiers using any valid Unicode character, even if that character doesn't exist on your keyboard or can't be displayed in the font your editor is using.

Verbatim string literals begin with @" and end with the matching quote. They do not have escape sequences, so

```
@"\\machine\share\path1\path2\file.ext"
is the same as:
"\\\\machine\\share\\path1\\path2\\file.ext"
```

is much simpler and less error-prone. Verbatim string literals also can extend over a line break. If they do, they include white spaces between the quotes:

```
@"First \t line
   tabbed second line"
// same as "First \\t line\n\ttabbed second line"
```

The only exception to the 'no escape sequence' rule for verbatim string literals is that you can put a double quotion mark inside a verbatim string by doubling it:

```
@"This is a ""quote"" from me." // same as "This is a \"quote\" from
me."
```

C# supports the following string operations in the language:

1. Indexing a string to read (but not write) the individual characters, as in s[i]
2. Concatenating two strings with the + operator, as in s + t
3. Equality and inequality, as in s == t and s != t (These operators will be added to the String class for Beta 2 and beyond and so, they can be used from any language that supports overloaded operators.) Concatenation will be done at compile time, if possible. You can also concatenate any object to a string—doing so concatenates the return value from the object's ToString method. For instance, all of the following are valid:

```
String s = " Hello "; String s1 = s + 5;    // " Hello 5"
String s2 = 6.3 + s1;   // "6.3 Hello 5"
```

In addition, because strings are immutable, the compiler and runtime will merge duplicate string literals so that there's only one copy of each literal in your program.

String literals and operations in Visual Basic.NET Visual Basic.NET string literals are very simple: They consist of a double quotation mark followed by a set of Unicode characters followed by a closing double-quotation mark. To include a double quotation mark within the string, double them, as in the following example:

```
"This prints ""Hello, world!"" on the screen"
```

There is no escape sequence mechanism for Visual Basic.NET strings or character literals except for the doubling of the double-quotation mark. Visual Basic.NET has a few string comparison operators that C# doesn't. These are:

1. Concatenation with String or Char using the + operator, as in s +

```
'c'
```

2. Concatenation with two Strings only using the & operator, as in s

```
& "There"
```

A complete set of comparisons: =, <>, >, <, >=, <=, which do either binary or text (culture-sensitive) comparison, depending on compilation option and the Option Compare statement Creating String objects In both languages, you can create a String object with the new statement. Most of the safe String constructors take an array of characters, so unless you have an array of characters, you will probably use another method to create your strings. (However, there is a constructor that takes a Char and an integer count—the resulting string will have the character repeated as many times as the count specifies.)
The most common methods will be initializing with a string literal and assigning a value from another string:

```
String s = "Hello, world!";
String t = s;   // no copy; s and t refer to same String
```

12.13.4 When is Clone not Clone?

But what if you want to have two separate strings, each containing the same value? Generally, this is not needed because strings are immutable, and hence there is not much point in having two separate strings that have the same value.

So, although String implements IClonable, String.Clone simply returns a reference to the same string without cloning it.

However, you can use the static method Copy if you want a second copy of the string:

```
String s = "Hello"; String t = (String)s.Clone();   // no copy; s and t
refer to same  String String u = String.Copy(s); // makes copy; s and u
refer to diff.  objects Console.WriteLine("s same as t: {0}, s same as
u: {1}",
Object)s == (Object)t, (Object)s == (Object)u);
```

We can also copy some or all of a string into a character array with the String.CopyTo method. Other interfaces, plus properties and a field, as we have seen in a String, implements ICloneable, although it does it in an odd (but reasonable) way. It also implements two other interfaces: IComparable and IConvertable. That means we can compare two strings using IComparable's CompareTo method (more on this later) and convert a string to all of the of built-in value types using the To—methods from IConvertable (more on this later, too).

String also has a few interesting properties. You can find the length (in characters, not bytes) of any string by accessing its Length property (which is read-only). And you can access the string one character at a time through the indexer property Chars (which is mapped to the C# indexer, and is also read-only):

```
String s = "Hello, world!";
Console.WriteLine("Length of \"Hello, world!\" is {0}",  s.Length); /
/ 13
for (int i = 0; i < s.Length; i++)
   Console.Write("{0} ", s[i]);
```

There is a static field called Empty, which contains the empty string—this gives you a language-independent way of expressing an empty string ("").

12.13.5 String equality/comparison methods

There are a variety of ways to compare two strings. For equality and inequality one major difference, of course, is whether the comparison is done by reference (the two strings point to the same object) or by value (they contain the same characters).

For both equality and relational comparisons, the other major difference is whether to use the current culture's collating order or the raw ordinal values for each character in the string. (A minor difference is whether the comparison is case-sensitive or not.) The default for comparisons is that they use the current culture of the thread they're running on and that they are case-sensitive. This is generally what you want.

The == operator generates a call to String.Equals, which does a culture-sensitive, case-sensitive comparison. If you want to do a reference comparison, cast both of the string references to Object, as in:

```
String s = "Hello", t = "there";
Boolean valueComp = (s == t);
Boolean refComp = ((Object)s == (Object)t); // In beta 2 and later,
use: // Boolean refComp = Object.ReferenceEquals(s, t);
```

Visual Basic.NET's equality, inequality, and comparison operators (=, <>, >, <, >=, <=) do either ordinal comparisons or culture- and case-sensitive comparisons, depending on the compiler options and the OPTION COMPARE statement. To do a reference comparison, use the IS operator. Visual Basic.NET also has a LIKE operator for string comparisons that does pattern matching.

The Equals method comes in several flavors. There are two instance methods: one takes a String as its parameter, and the other takes an Object (which must refer to a String). There is also a static version of Equals, which takes two strings. The version that takes an Object is overridden (originally defined in Object), the one that takes a string is faster because no conversion is necessary, and the static version can handle being passed null without throwing an exception.

For comparisons, there are three methods: Compare, CompareOrdinal, and CompareTo. All of these methods return a negative number if the first string is less than the second, zero if they have the same value, and a positive number if the first string is greater than the second.

Compare is a set of overloaded static methods that do culture-sensitive comparisons, case-sensitive by default. Each of the overloads takes at least the two strings to compare; there are also overloads to take a Boolean that specifies case sensitivity and Int32s to specify indices and length of substrings to compare (rather than comparing the whole string).

CompareOrdinal is a pair of overloaded static methods that do ordinal comparisons of two strings or substrings within the two strings.

CompareTo is an instance method that implements IComparable. It compares the current string with the string or object passed as a parameter using culture-sensitive and case-sensitive comparisons.

Searching a string: EndsWith/StartsWith/IndexOf/LastIndexOf/Substring. There are a bunch of methods for figuring out what's in a string. EndsWith and StartsWith return true if the current string ends/starts with the specified string, or false otherwise.

IndexOf and LastIndexOf each have a bunch of overloads. Each returns the first/last position in the current string (or a substring of it) at which the string (or character array) occurs.

Substring is like the old GW-BASIC MID$ function. There are two overloads: One creates and returns a new string that contains the substring from the specified index to the end of the string, and the other creates and returns a new string that contains the substring of the specified length starting at the specified index.

Formatting

The String class has a family of static Format method overloads. All of them provide formatting abilities the same as the ones we've been using in our calls to Console.WriteLine by creating and returning a new string with the format specifier replaced by the string representation of the remaining parameters. For instance, you could use:

```
String f = String.Format("The value is {0}", 5);
Console.WriteLine(f); // prints "The value is 5" (without quotes)
```

There are other ways to format data as well, and a lot of issues around culture awareness. Unfortunately, we don't have the time to go into them this time around.

Padding and trimming

There are a variety of methods for padding the left or right part of a string with blanks (or any character you like) called PadLeft and PadRight; and for trimming the blanks or any set of characters you specify off the beginning and/or end of a string (TrimStart, TrimEnd, and Trim). These all create new strings and return them.

Insert/Remove/Replace/Concat

These methods are somewhat similar to the ones found in StringBuilder, although there are fewer overloads of them in String than in StringBuilder. See the section to follow on StringBuilder for more on the differences.

Insert creates a new string with the specified string inserted at a position specified. Remove creates a new string with the specified number of characters removed from the specified position. Replace creates and returns a new string with all occurrences of a specified character replaced by another character. And the various overloads of the static method Concat create and return new strings by concatenating the strings and/or objects it's passed.

Split/Join

Split and Join are powerful methods. Join is a static method that returns a string created by concatenating some or all of the strings in the string array it's passed, using a specified string as a separator. Split splits a string into a newly created and returned array of strings, using the specified set of separator characters.

Conversions

String contains a bunch of conversion methods, most of which convert to some built-in type and are of the form To. The list that converts to built-in types includes ToBoolean, ToByte, ToChar, ToDecimal, ToDouble, ToInt16, ToInt32, ToInt64, ToSByte, ToSingle, ToString, ToUInt16, ToUInt32, and ToUInt64. There are a couple of methods to convert to other types: ToCharArray and ToDateTime. And there are methods to convert the string to upper- or lower-case: ToUpper

and ToLower. Finally, ToType converts to the type specified by the Type object passed as its parameter.

12.13.6 Intern/IsInterned

The .NET Runtime maintains a pool of literal strings in the application domain (roughly similar to a process). When it loads an assembly, such as your program, into an application domain, it merges the assembly's string literals into the application domain's pool of string literals, thereby avoiding duplication of string literals. (This can be done without changing your program's results because strings are immutable.) The fact that all string literals are interned also means that you can make reference comparisons of string literals correctly, since there's only one copy of each string literal.

However, if you build a string yourself, such as through one of the String methods that creates a new string or by using a StringBuilder, it will be a distinct string from the string with the same value in the literal pool (assuming there is one), so reference comparisons will fail. Reference comparisons are much faster than value comparisons—all that has to be compared is the address.

If you want to add your strings into the literal pool, you can do so with the static method Intern, which adds the string to the literal pool (if it's not there already) and returns a reference to the literal pool string. For instance:

Listing 12.8

```
String s2 = new
StringBuilder().Append("Foo").Append("Bar").ToString(); String s3
=String.Intern(s2);   // return reference to literal  pool
String s = "FooBar";              // was in literal pool all along
Console.WriteLine((Object)s2==(Object)s);  // false: different
Console.WriteLine((Object)s3==(Object)s);   // true: same
// Beta2 and later: use
Object.ReferenceEquals
```

You can also check to see if a string exists in the pool already with the static IsInterned method, which returns true if the string is in the pool.

GetHashCode: Please recall that if you override Equals, you should also override GetHashCode. In String, Equals is overridden, so GetHashCode is as well, and provides good hashing and good performance.

StringBuilder: You may have noticed that many of the methods in String create and return new strings. This can be expensive if you allocate and throw away many strings.

As a rule, if you are going to do only one string operation on a particular string that involves creating a new string, go ahead and use the appropriate method in String. But if you are doing more than one, and if the operations you need are available in System.Text.Stringbuilder, create a StringBuilder from your string, do the multiple operations on the StringBuilder, and then call ToString on the StringBuilder to get the result string back. See the preceding example in which this is done all in one line.

It is very common to build a string in a StringBuilder, manipulate it, and convert it to a string. Note that the call to ToString won't actually copy the string unless you later modify the same StringBuilder object, so it's actually very efficient—and the overhead for creating the StringBuilder is one copy operation, not two.

You may also want to use a StringBuilder if the overloads available are considerably easier to use than the more limited overloads in String. For instance, StringBuilder has a Replace method overload that can replace (and even delete) entire strings, not just replace single characters as in the single overload of String.Replace. If you need to replace all of the occurrences of a substring you will probably want to use a StringBuilder, even though you only do one (very tedious) operation.

To use StringBuilder you'll need to include 'using System.Text;' at the beginning of your program file.

StringBuilder has several constructors that can initialize the object from a string and set its capacity and maximum capacity. (The default maximum capacity is about two billion characters, so it's not unreasonable to set it to a somewhat smaller value.) The capacity of the object will be adjusted upwards as needed, as long as you don't exceed the maximum capacity. The maximum capacity cannot be adjusted after the StringBuilder is constructed.

There are four properties: Capacity, which can be read or written; MaxCapacity, which is read-only; Length, the current length of the string (can be set shorter or longer); and Chars, an indexer that allows you to read and write individual characters. The EnsureCapacity method allows you to increase capacity if it's not already large enough.

The Append method allows you to append a string or any type (there are many overloads) to the end of the StringBuilder. If you pass another type, its ToString method will be called and the result appended to the StringBuilder.

The AppendFormat method allows you to format the string you append to the StringBuilder. The formatting is the same as String.Format and Console.WriteLine.

The many overloads of Insert allow you to insert a string (perhaps calculated from a call to your parameter's ToString method) anywhere in the StringBuilder.

Remove allows you to remove any number of characters from any position.

Replace allows replacement of individual characters or, as just mentioned, substrings.

Finally, the overloads of ToString create and return a new String object from the StringBuilder (or the specified substring of it).

System.Text Encoders and Decoders You've noticed that all of the strings in a .NET Framework program are stored as 16-bit Unicode. You may also have noticed that not everyone uses Unicode, so sometimes you'll have to convert from some other character encoding to Unicode, or from Unicode to some other character encoding.

The .NET Framework provides several classes to do encoding (converting Unicode characters to a block of bytes in another encoding) and decoding (converting a block of bytes in some encoding to Unicode characters).

There is a class for each supported encoding: ASCIIEncoding, CodePageEncoding, UnicodeEncoding (useful for converting from big-endian to little-endian), UTF7Encoding, and UTF8Encoding.

Each of these classes has methods for both encoding (such as GetBytes) and decoding (such as GetChars) for encoding and decoding a single array all at once. In addition, each supports GetEncoder and GetDecoder, which return encoders and decoders capable of maintaining shift state so they can be used with streams and blocks.

Regular Expressions You may have noticed that Visual Basic.NET supports limited regular expression matching via the LIKE statement. In addition, the .NET Framework supports some very sophisticated regular expression processing in the System.Text.RegularExpressions namespace.The keyclass here is System.Text.RegularExpressions.RegEx, which represents a compiled regular expression and provides methods for using it, including finding and replacing regular expressions in strings and splitting a string into an array of strings. The documentation on regular expressions is sparse now, so we won't go into it yet. Perhaps in a future column.

Formatting The basic story is this: It is possible to put things other than the index number in the braces inside the formatting string in Console.WriteLine (and String.Format, which is called by Console.WriteLine). The full form of the format specifier is:

```
{index [, width][:formatstring]}
```

where index is the zero-based number of the parameter after the formatting string to which this formatter refers; width, if present, is the width (in spaces) of the field to format into. A positive number for width means that the result is to be formatted right-justified; a negative number means that the number is to be formatted left-justified in the field. (See the first two examples, to follow.)
The optional formatstring contains formatting instructions specific to the type being formatted. If the object implements IFormattable, formatstring is passed to the object's Format method (the siqnature of this method changes to ToString(string, IFormatProvider) in Beta 2 and after, but the functionality will be the same). If the object does not implement IFormattable, Object.ToString() will be called instead and the formatstring will be ignored.

Note also that ToString, which is not sensitive to the current culture in Beta 1, will be culture-sensitive in Beta 2 and beyond. For instance, 1,234.56 would format as 1.234,56 in countries where "." is used to separate thousands and "," is the decimal separator If you need results to be exactly the same regardless of culture, use CultureInfo.InvariantCulture as the culture.

For the complete story on formatting, check out the Formatting Overview in the .NET Framework Developer's Guide.

Number Formatting Note that numbers are formatted in a culture-sensitive way: The separators and spacing between separators, as well as the currency sign, are all determined by the culture—by default, the default culture on your machine. The default culture is associated with your thread of execution, and you can get and set that culture with Thread.CurrentThread.CurrentCulture. There are ways we won't go into just yet to change the culture for a just a given formatting operation.

Letter formats for built-in types One type of formatting command starts with a single letter out of the following set:

G—general, shorter of E or F

F—floating point, regular notation

E—floating point in E notation (where E stands for 'times 10 to the')

N—floating point with separators (commas in the U.S.)

C—currency, with currency sign and separators (commas in the U.S.)

D—decimal, for integral types only

X—hexadecimal, for integral types only

The letter may be followed by a number that indicates, depending on the letter before, the total number of digits, the total number of significant digits, or the number of digits after the decimal point.

Here are some examples of letter formats:

Listing 12.9

```
double pi = Math.PI; double p0 = pi * 10000; int i = 123;
Console.WriteLine("Floating-point formatting, no separators
(commas)"); Console.WriteLine("pi, Left {0, -25}",   pi); //
3.1415926535897931 Console.WriteLine("p0, Rt.  {0, 25}",    p0); //
3.1415926535897931
Console.WriteLine("pi, E    {0, 25:E}",  pi); // 3.1416E+000
Console.WriteLine("In E and F, n (here, 4) digits AFTER the
decimal"); Console.WriteLine("pi, E4   {0, 25:E4}", pi); //
3.1416E+000
Console.WriteLine("pi, F4   {0, 25:F4}", pi); // 3.1416
Console.WriteLine("In G format, 4 SIGNIFICANT digits—using E if
needed"); Console.WriteLine("pi, G4   {0, 25:G4}", pi); // 3.142
Console.WriteLine("p0, G4   {0, 25:G4}", p0); // 3.142E4
Console.WriteLine("N and C formats have commas (separators), " +
          "default to two digits after decimal, rounded toward
even."); Console.WriteLine("p0, N    {0, 25:N}",  p0); // 31,415.93
Console.WriteLine("p0, N4   {0, 25:N4}", p0); // 31,415.9265
Console.WriteLine("p0, C    {0, 25:C}",   pi); // $3.14
Console.WriteLine("D and X are for integral types ONLY, " +
          "non-integral throw FormatException—X is hex");
Console.WriteLine("i,  D    {0, 25:D}",  i ); // 123
Console.WriteLine("i,  D7   {0, 25:D7}", i ); // 0000123
Console.WriteLine("i,  X    {0, 25:X}",  i ); // 7B
Console.WriteLine("i,  X8   {0, 25:X8}", i ); // 0000007B
```

12.14 Picture formats

Instead of a letter format, the formatstring can contain a picture format. There are a few examples in the code excerpt to follow. (This is similar to the Basic 'Print Using' statement.) The picture format capabilities even include the ability to format negative, positive, and zero numbers differently. There are a few picture format features not included in the example below. For more information, see the topic Picture Format Numeric String in the .NET Framework Developer's Guide or in the documentation.

In the following example, we have used both the "#" and "0" characters together. The "#" character is replaced with nothing if the corresponding digit is a leading or training zero. The "0" character is replaced with a zero character regardless of the value of the corresponding digit—so the number will be padded with zeroes. The period, if present, represents the position of the decimal separator.

So why use the characters together, as in "###0.##"? The "#" picture character is replaced with nothing (not even a zero character) if the value to be formatted is exactly zero. You will always want to include at least one "0" just to the left of the decimal point—otherwise, the field will produce no output if the value is zero. In other words, formats that contain only "#" characters without at least one "0" are almost always a programming error.

The comma has two uses: If a comma or set of commas is immediately to the left of the period (or at the end if no period), they tell the formatter to divide the number printed by 10 ** (3 * n), where n is the number of commas. In other words, the number is divided by thousands or millions or billions or even more.

If the comma is to the left of at least one "0" or "#" placeholder, it tells the formatter to place the appropriate group separator character (in the U.S., a comma) between groups of digits. (In the U.S., the groups are three digits each.)

You can format percentages by putting a "%" in your picture. The "%" will print in the position you specify and the number will be multiplied by 100 before printing (so 0.28 becomes 28%).

If you want to use picture formatting for exponential notation, you can specify an "e" or "E" followed by a plus or minus sign followed by any number of zeroes, as in "E+00" or "e-000". If you use "e" a lowercase "e" will be printed. If you use "E" an uppercase "E" will be printed. If you use a plus sign the sign of the exponent will always be present. If you use a minus sign the sign will only be present if negative. (There's a bug in the handling of "-" in Beta 1 that causes the negative sign to always appear.)

There is also conditional formatting, based on the sign of the number being formatted. Just include two or three separate formats separated by a semicolon in your format string. If you have two, the first will be for non-negative numbers and the second for negative numbers. If you have three the first will be used for positive numbers, the second for negative, and the third for zero.

You can include literal characters in format strings. If the character you want has a special meaning, use a backslash in front of it to "escape" it. For instance, if you want to print a percent sign without causing the multiplication by 100, you could use a backslash in front of it (you will have to double the backslashes in C++ and C#), as in "#0.##\\%"(If you're using C#'s, you may want to use the incredibly cool verbatim sting literals, as in @"#0.##\%"). Or you can enclose a string in single or double quotation marks to avoid having its characters interpreted as formatting commands. In Beta 2 and later, you can include literal braces in a formatting string by doubling them.

Here are some examples of picture formats:

Listing 12.10

```
long m34 = 34000000; // 34,000,000
Console.WriteLine("A few picture formats");
Console.WriteLine("0 prints zero if no digit; " +
```

```
        "literals such as i: always print");
Console.WriteLine("\tperiod represents position of decimal
separator");
Console.WriteLine("i,  i: 0000.0 {0, 10:i: 0000.0}",  i); //
i:0123.0 Console.WriteLine("# disappears if no significant digit, " +
        "a comma means to put separators in");
Console.WriteLine("Be SURE to use at least one 0 in numeric
pictures."); Console.WriteLine("p0, ##,##0.#  {0, 10:##,##0.#}",-p0);
//   - 31,415.9 Console.WriteLine("m34, 0,,      {0, 10:0,,
million}", m34); //  34 million Console.WriteLine("p0, #0.#E+00  {0,
10:#0.#E+00}", p0); //     31.4E+03 Console.WriteLine("% multiplies
by 100 and prints percent sign"); Console.WriteLine("pi, ###0.##%
{0, 10:###0.##%}", pi); //      314.16% Console.WriteLine("No
multiplication because of \\ " + "(Note: backslashes doubled!)");
Console.WriteLine("pi, ###0.##\\\\% {0, 10:###0.##\\%}", pi); //
3.14% Console.WriteLine("Same with C#'s verbatim string");
Console.WriteLine(@"pi, ###0.##\\% {0, 10:###0.##\%}", pi); //
3.14% Console.WriteLine("10, '#'#0     {0, 10:'#'#0}",      10); //
#10
Console.WriteLine("Conditional formatting based on sign");
Console.WriteLine("# if zero/pos, (#) if negative");
Console.WriteLine("-5  0;(0)     {0, 10:0;(0)}",    -5); // (5)
Console.WriteLine("# if pos, -# if neg, zip if zero");
Console.WriteLine(" 0  0;-0;zip  {0, 10:0;-0;zip}",  0); // zip
```

As you can see, the formatting capabilities are very powerful.

12.14.1 How Formatting Works

The examples in the documentation call the Format method on variables of the passed object's type. These Format methods are passed just the formatstring part of the formatting specification, but are not passed the index, nor the width. (The call to Format will change to ToString in Beta 2.)

The index and width are used by String.Format (which is called by Console.Write and Console.WriteLine) to get the proper object on which to call Format and to left- or right-justify the result from that call. (By the way, if the object being formatted doesn't implement IFormattable (and therefore, the Format method), String.Format will call the object's ToString() method instead, ignoring formatstring.)

In other words, Console.WriteLine calls String.Format, passing all of the parameters it was passed. String.Format parses the string, looking for "{" characters. When it finds one, it parses the substring up to the first "}" to determine the index number, width, and formatstring. It then accesses the appropriate parameter by index and calls its Format method, passing the formatstring part of the "{}" section. (If the parameter object doesn't implement IFormattable, ToString is called.)

In either event, a string is returned and String.Format concatenates it to the result string before continuing to parse the format string. When it's done, String.Format returns the resulting formatted string to Console.WriteLine, which prints it.

For Beta 2 and later, the object's Format method (which is the Format method in IFormattable) is replaced by an override of ToString that takes a formatting string and an IFormatProvider (or null). But String.Format will still exist, so those calls won't change.

Customizing formatting You can even write your own formatters, either for your own types or as custom formatters for built-in types, as explained in Customizing the Format Method in the .NET Framework Developer's Guide. If you write a custom formatter for a built-in type, you can't use it from Console.WriteLine, but you can use it by calling the overload of String.Format that takes an IServiceObjectProvider (called IFormatProvider in beta 2 and later) as a parameter.

Date and Time Formatting You will recall that there is a class called DateTime that can hold a date and time. As you might guess, there are a wide variety of ways to format DateTime objects: just the date, just the time, in universal or local time, with day/month/year in several orders, even sortable. Date and time formatting is culture-sensitive.

You can also use a custom format string to format DateTime objects. Such strings would contain case-sensitive substrings of certain letters to represent various parts of the date and time, such as day of the week, day, month, year, era, hour, minute, second, or time zone. Many of these parts have various formats: For instance, M is numeric month without leading zero, MM is numeric with leading zero, MMM is the three-letter abbreviation, and MMMM is the complete month name spelled out in the language appropriate for your current culture. The complete list of both custom and standard formatting characters can be found in the .NET Framework Reference.

Here is an example of date and time formatting:

Listing 12.11

```
Console.WriteLine("Standard formats"); // more on Parse later
DateTime dt = DateTime.Parse("1 Jan 2001 12:01:00am");
Console.WriteLine("d:  {0:d}", dt); // 1/1/2001
Console.WriteLine("D:  {0:D}", dt); // Monday, January 01, 2001
Console.WriteLine("f:  {0:f}", dt); // Monday, January 01, 2001
12:01 AM Console.Write("F:  {0:F}", dt); // Monday, January 01, 2001
12:01:00 AM Console.WriteLine();
Console.WriteLine("g:  {0:g}", dt); // 1/1/2001 12:01 AM
Console.WriteLine("G:  {0:G}", dt); // 1/1/2001 12:01:00 AM
Console.WriteLine("M/m: {0:M}", dt); // January 01
Console.WriteLine("R/r: {0:R}", dt); // Mon, 01 Jan 2001 08:01:00
GMT Console.WriteLine("s:  {0:s}", dt); // 2001-01-01T00:01:00
Console.WriteLine("t:  {0:t}", dt); // 12:01 AM
Console.WriteLine("T:  {0:T}", dt); // 12:01:00 AM
Console.WriteLine("u:  {0:u}", dt); // 2001-01-01 08:01:00Z
Console.Write("U:  {0:U}", dt); // Monday, January 01, 2001 8:01:00
AM Console.WriteLine();
Console.WriteLine("Y/y: {0:Y}", dt);
```

```
// January, 2001
Console.WriteLine("Custom formats");
// have to "escape" characters used as formats_in this case, t  and z
// using both quote marks (in literal strings) and backslashes
Console.WriteLine(@"dddd, dd MMMM yyyy"" at ""HH:mm:ss in \zone
zzz:"); Console.WriteLine(@"{0:dddd, dd MMMM yyyy"" at ""HH:mm:ss in
\zone zzz}",
      dt);
   //    Monday, 01 January 2001 at 00:01:00 in zone -08:00
```

Parsing: Converting from a String to a Number/Date Each of the numeric classes and the DateTime class has one or more static Parse methods. The simplest of these just takes a string. The numeric classes have an overload that also takes a NumberStyles enumeration containing flags describing which non-numeric characters are allowed when parsing. (There is more information on NumberStyles enumeration in the .NET Framework Reference.)
It is also possible to use a culture other than the default culture for parsing, as well as formatting.

So, for instance, to parse a string into a double, you may write:

Listing 12.12

```
String s1 = "123.450";   // could have gotten string from
Console  or file Double d1 = Double.Parse(s1);
Console.WriteLine("Result of parsing \"123.450\" is {0}", d1);
String s2 = "$1,234.56";
Console.WriteLine("We'll get an exception if we don't pass the " +
"right NumberStyles");
Double d2 = Double.Parse(s2, NumberStyles.Currency);
Console.WriteLine("Result of parsing \"$1,234.56\" with " +
"NumberStyles.Currency is {0}", d2);
```

Note that many classes in Beta 1 have a FromString method. This method will be removed from most classes in Beta 2 and later. Use Parse instead and save yourself the trouble of updating your code.
Collections Overview An array is a simple form of what are known in programming as collections. A collection is just what it sounds like: a group (or collection) of objects. Different types of collections will have different methods of storage and access—and different performance characteristics in various situations.

For instance, an array is stored contiguously in memory. That means indexing it (accessing the ith element) is very fast. But inserting an element into the middle of an array is slow because, at minimum, you have to move all of the elements in the array after the element you insert. (At worst, you will have no space in the array and will have to create a new one and copy all the elements into it.) Searching for something in an array can be fast if the contents of the array are sorted (you can use a binary search), but slow if they're not (because you'll have to look at each element in turn).

A hash table stores references to values, each associated with a key. From a logical standpoint, you can think of the keys of a hash table as being like array subscripts: When you present the key to the hash table, you get back the object that's associated with that key. (Obviously, keys

must be unique, just like array subscripts.) In fact, many languages, including C#, use array subscript syntax for lookup, as in hashTab["key value"]. The key can be of any type that properly implements Object.GetHashCode and Object.Equals, not just String or Int32.

The values are not stored contiguously, and you can't step through a hash table with an integer index (unless your keys are contiguous integers). However, you can enumerate all of the values using foreach. Adding and deleting elements is pretty fast, as is element lookup by key.

The sorted list type combines an array of keys with an array of reference to objects to give you a sorted array that you can both index and access by key. It is not as fast as a hash table, but it is a good choice when you need indexing as well as lookup.

You can combine collections to get the performance characteristics you need. For instance, you could use a hash table that points to the elements of a list or array to do key lookups in a list or array quickly. You'd have to be sure to keep the hash table in sync with the list or array, though. It would be good to write a collection that contains and properly maintains both containers.

If you want to access items in the same order in which they were added to the collection, use a queue, which is optimized for first in, first out access. If you want to access the items in the opposite order (last in, first out), use a stack.

Many systems also have linked list types, although the .NET Framework doesn't. Linked lists are good when you need to add and delete items often, because once you find the proper position, you add an item by switching a few pointers. And it's fast to print items out in order, although indexing and searching are slow. In most cases, one of the list types based on arrays works as well or better than a linked list, although there are some cases where you need to add and delete items from the middle of a list frequently, for which a linked list might be better.

The common tree structure is also missing from the .NET Framework. It's reasonably fast at most operations—adding, deleting, and searching—although it can't be indexed efficiently. However, trees can be traversed in order efficiently. In most cases, a hash table or a sorted list works as well or better than a tree, although a hash table doesn't keep the items in order and a sorted list is slower when adding and deleting items.

NET Framework Collections: Basic Interfaces In order to understand the .NET Framework collections, it is helpful to understand some of the interfaces that almost all of the collections implement.

IEnumerable: All of the collections implement at least two interfaces: ICollection and IEnumerable.

IEnumerable has a single method called GetEnumerator. It returns an enumerator object that implements the IEnumerator interface.

The foreach statement in C# and Visual Basic.NET relies on the collection implementing IEnumerable.

IEnumerator: The IEnumerator interface allows you to enumerate a collection—in otherwords, visit every element. This visit will be in order if the collection is an ordered collection such as an array or list, in no particular order for unordered collections such as hash tables. IEnumerator has a read-only property called Current, which returns a reference to the current object. It alsohas two methods: MoveNext, which moves to the next object in the collection, sets Current to point to that object, and returns true if it's not gone past the end; and Reset, which resets the

Current pointer to an undefined value before the beginning of the collection. Before you first access Current after creating the enumerator or just after calling Reset, you must call MoveNext because the Current reference is initially undefined.

Note that collections do not themselves implement IEnumerator; rather, they return a separate enumerator object each time GetEnumerator is called. A collection can be enumerated both by using foreach and by using the enumeration interfaces directly.

ICollection: ICollection extends IEnumerable and contains an additional three read-only properties and a method. The properties are Count, which returns the number of objects in the collection; IsSynchronized, which returns true if access to the collection is synchronized for multithreaded access; and SyncRoot, which returns an object you can use to synchronize access to the collection across threads (if it's not already synchronized by design). The method is CopyTo, which copies elements from the collection into an array, starting at the array position specified.

Using the ICollection and IEnumerable Every collection can tell you how many elements it has, can be enumerated and copied, and is capable of being synchronized. So, you could pass any collection in to the following methods:

Listing 12.13

```
public static void PrintCollection(String info, ICollection c) {
   Console.Write("{0}: Collection has {1} elements: ", info,
c.Count);
   foreach (Object o in c) { // simplest way to write loop
      Console.Write("{0} ", o);
   }
   Console.WriteLine();
}
public static void PrintCollectionSync(ICollection c) {
   Console.WriteLine("Synchronized access (not necessary here)");
   lock (c.SyncRoot) {  // locking on SyncRoot
   //—just for demo, not necessary unless multi-thread access
   // loop below equivalent to foreach above
      IEnumerator e = c.GetEnumerator();
      while (e.MoveNext()) { // move FIRST
         Console.Write("{0} ", e.Current);
      }
      Console.WriteLine();
   }
}
public static Object [] CopyToArray(ICollection c) {
   Object [] retVal = new Object[c.Count];
   c.CopyTo(retVal, 0);
   return retVal;
}
```

Note that foreach is the preferred way to access the collection if your programming language supports it, as do C# and Visual Basic.NET. Unless you have a good reason not to, you should use foreach because using it allows the compiler to generate better code for specific collection accesses. This is true regardless of whether you use synchronization (we do in the second method above, but it's not necessary unless more than one thread can access the same collection at the same time).

In the following example, we do indeed pass in two different collections: an array of Int32, and an array of Objects. You can try these method calls with other collections on your own:

```
Int32 [] intArr = {0, 1, 2, 3, 4}; PrintCollection("intArr", intArr);
Object [] oArray = CollectionExample.CopyToArray(intArr);
PrintCollection("oArray (copied from intArr)", oArray);
PrintCollectionSync(oArray);
```

IList contains many of the collections—specifically, those that are ordered and therefore indexable also implements IList, which extends ICollection (which, as you will recall, extends IEnumerable) for indexable ordered lists. IList adds one property (three properties in Beta 2 and later) and seven methods.

The property is Item, which takes an index and returns the corresponding element of the list. (Item is the indexer for C#, so the syntax for accessing the Item property in C# for the collection Foo is Foo[indexer].) The Beta 2 and later properties are IsFixedSize and IsReadOnly, which return true if the list is of fixed size or is read-only, respectively.

There are a bunch of methods in Ilist: Clear clears the list; Add adds an item to the list, typically at the end (depending on the actual implementation); Insert inserts an item in the list at the specified index; Remove removes the first instance of a particular object and RemoveAt removes the object at a specified index; Contains tells whether the list contains a particular value; and IndexOf returns the index of a value's position in the list. Note that you can't use Add, Insert, Remove, or RemoveAt on fixed-size collections, such as arrays. Attempting to do so will throw an exception.

But we can create an ArrayList from an array easily—and we can demonstrate the IList methods with that. To create the ArrayList and make the call, the code is simple:

ArrayList al = new ArrayList(intArr); ListTest(al);

The ListTest method is simple:

Listing 12.14

```
public static void ListTest(IList l) {
   PrintCollection("Original list", l);
   l.Add(5);
   PrintCollection("After Add(5)", l);
   l.RemoveAt(2);
   PrintCollection("After RemoveAt(2)", l);
   l.Remove(3);
   PrintCollection("After Remove(3)", l);
   l.Insert(1, 7);
   PrintCollection("After Insert(1, 7)", l);
```

```
        Console.WriteLine("Contains 55? {0}", l.Contains(55));
        Console.WriteLine("IndexOf(4) is {0}", l.IndexOf(4));
        l.Clear();
        PrintCollection("After clearing", l);
    }
```

The output in this case is as follows:

```
Original list: Collection has 5 elements: 0 1 2 3 4 After Add(5):
Collection has 6 elements: 0 1 2 3 4 5 After RemoveAt(2): Collection has
5 elements: 0 1 3 4 5 After Remove(3): Collection has 4 elements: 0 1 4
5 After Insert(1, 7): Collection has 5 elements: 0 7 1 4 5 Contains 55?
False IndexOf(4) is 3 After clearing: Collection has 0 elements:
```

http://www.dmh2000.com/ David M. Howard Microsoft Certified Solution Developer Sun Microsystems Certified Java Developer

12.14.2 Formatted Strings in .NET

Suppose you have some strings and some numbers and you want to print them out to the console in a specified format. In C, you use the printf family. It is easy, but error prone and not type safe.

In C++, you use output streams, the insertion operator (<<) and manipulators. Type is safe at compile time but very complicated to use.

In VB, you use the Format function, which lets you apply a 'mask' to a number to produce a particular formatted output. It is easy and safe, in the sense that if the arguments don't match up it will take defaults without crashing.

In .NET, the VB methodology is mixed in with an approximation of C type variable argument lists to fit C# syntax and semantics. Its easy to use and runtime type safe (throws exceptions if types don't match up).

The .NET scheme has a lot in common with the format schemes from java.Text, with a lot less complexity and a little more help to build up formatted strings in a parameterized way.

12.15 STRING FORMATTING IN .NET

The NGWSSDK QuickStart samples have an QuickStart sample program that shows how to use most of the formatting. You can also search for 'User Defined Numeric Formats' and 'Named Numeric Formats' in either the NGWSSDK or MSDN documentation.

In the .NET framework, the System.String object has a set of overloaded methods named 'Format'. The syntax and semantics are a little different from VB. The examples from here on are in C#, but the VB code is about the same minus the semicolons. The examples below are contained in the file examples.cs

The simplest System.String.Format method signature is:

```
public static string System.String.Format(string format,object  arg0);
```

Let us see how it works.
The 'format' string argument provides a template into which the value of the arg0 object is plugged into. The template will consist of a string with [optional] format specifiers embedded in it.

```
Example 1 - no formatting instructions: string s =
System.String.Format("foo bar.","from me");
result s == "foo bar."
```

The arg0 argument is not applied because there are no format specifiers embedded in the format string. According to the current Framework documentation, arg0 can be null, however in practice that causes an FormatException to be thrown.

```
Example 2 - insert arg0 into string string s = System.String.Format("foo
bar {0}.","baz quux");
result s == "foo bar bax quux."
```

The arg0 argument is inserted at the location of the "{0}" format specifier.
The format specifiers are of the form "{N[,M]:FormatString}" N := argument number starting with 0 M := integer value specifying field width and alignment positive M is right justified,negative M is left justified, magnitude of M specified field width. FormatString := type dependent formatting template (we will get to this in a minute)

If you don't specify a field width 'M' and/or FormatString then you get the exact width of the string produced by the argument.

Example 3 - insert arg0 into string in a field width of 10, right justified

```
string s = System.String.Format("foo bar {0,10}.","baz quux");
result s == "foo bar baz quux."
```

Example 4 - insert arg0 into string in a field width of 10, left justified

```
string s = System.String.Format("foo bar {0,-10}.","baz quux");
result s == "foo bar baz quux ."
```

Because there are several overloads of the Format function, you can have up to 4 explicit arguments or an array of arguments. This lets you format a string with various combinations of multiple elements. If you have more arguments listed than format specifiers in the string, it acts like printf and just ignores the extraneous arguments. However, if you have more format specifiers than arguments, you will get a FormatException.

Example 5 - insert arg0,arg1 and arg2

```
string s = System.String.Format("foo {0} {1}
                  {2}.","bar","baz","quux");
result s == "foo bar baz quux."
```

Example 6 - more arguments than format specifiers (prints "foo bar")

```
string s = System.String.Format("foo {0}.","bar","baz","quux");
result s == "foo bar ."
```

Example 7 - more format specifiers than arguments (throws FormatException)

```
string s = System.String.Format("foo {0} {1} {2}.","bar");
```

```
result == FormatException "The input string wasn't in a correct
format"
```

So far we have just used other strings as the arguments to Format. Actually, any type can be used. Usually, if you specify some type other than string as an argument, you get the string produced by the argn.ToString method (there is an interesting exception to this rule that will be covered in a moment). Because in the method signatures for Format the arguments are Object's, if you specify a simple type or value type you will get a boxed version of that type. The boxed version will have a default 'ToString' that is used to produce the output string.
Using strings as arguments just lets you plug in the string with a field width and alignment specification. With numeric data types, it is slightly different. This is where the FormatString portion of the format specifier comes into play.

Before getting into the formats, let us not forget that all formatting is subject to localization rules and the default locale, so the actual output string may vary depending. The examples here use the default US English locale and numbering schemes. Note that the FormatString is specified in a locale independent way, it is just the output that can vary.

12.16 User Defined Numeric Formats

The kinds of things we will want to do with numeric values include specifying precision, width, decimal point, thousands separator, leading zero's, scientific format etc. For this chapter, we will use only 3 of the special characters.

0 : digit placeholder (significant digit or 0) # : digit placeholder (significant digit or nothing (no character)) . : decimal placeholder (will generate locale specific output)

Some more examples are as follows:

Example : insert a number in default format

```
string s = System.String.Format("foo {0}",1);
result s == "foo 1"
```

Example : insert a number with leading zeros

```
string s = System.String.Format("foo {0:0000}",1);
result s == "foo 0001"
```

Example : insert a number

```
string s = System.String.Format("foo {0:####}",1);
result s == "foo 1"
```

If you specify a numeric format with '0's and/or '#'s, when the string representation of the number has more digits than you allowed for in the FormatString, the actual value will be presented (like printf with a field width).

> Note that in example 10, the # s don t produce leading spaces. If you want leading spaces, use the ',M field width specification. The # s are most useful for specifying values to right of the decimal point.

12.17 Floating Point Formats

Example : insert a number with decimal point and one decimal digit

```
string s = System.String.Format("foo {0:#.#}",21.123);
result s == "foo 1.1"
```

Example : insert a number with decimal point and two decimal digits

```
string s = System.String.Format("foo {0:#.##}",21.123);
result s == "foo 1.12"
```

Example : insert a number with decimal point and 3 decimal digits

```
string s = System.String.Format("foo {0:#.###}",21.123);
result s == "foo 21.123"
```

Example: insert a number with decimal point, 3 decimal digits and leading 0

```
string s = System.String.Format("foo {0:0.###}",0.123);
result s == "foo 0.123"
```

Example: type mismatch between format specifier and argument

```
string s = System.String.Format("foo {0:0.###}","xyz");
result == FormatException "The input string wasn't in a correct format"
```

12.17.1 WriteLine

The next thing to understand is that the the System.IO.TextWriter class, which defines the WriteLine methods used by StreamWriter and Console, has overloads that implement the same syntax and semantics as System.String.Format. For example, the TextWriter method:

```
public virtual void WriteLine(String, Object);
```

writes out a formatted string and a new line, using the same semantics as Format.
Further, there are WriteLine overloads that match most of the System.String.Format signatures, except that you can only go up to 3 Object arguments in WriteLine instead of the 4 provided for in System.String.Format. Go Figure.

You do not have to separately Format a string and the Write it. Just use WriteLine as you would Format.

Example : insert arg0,arg1 and arg2 with WriteLine instead of Format

```
Console.WriteLine("foo {0} {1} {2}.","bar","baz","quux");
```

12.18 Predefined Formats

There is also a set of predefined formats that are indicated by using the appropriate FormatString alpha character to specify which predefined format to use. These formats echo what is available in VB and by extension most of the Microsoft Office products such as Excel. If you do a 'format cell' in Excel, you will see some of these formats in the list of what is available.

{0:g} - General Number

{0:e} - Scientific (has several options, see documentation)

{0:f} - Fixed

{0:c} - Currency

0:d} - Decimal

0:x} - Hex

12.19 Polymorphic Formatting

What happens if we want to insert string representations of arbitrary objects into a string, using System.String.Format? Let us consider this now.

The original motivation for this section is that when we are programming, we are likely to come across where we want to print an integer in hexadecimal. The standard format {0:x} that prints a hex number doesn't support the picture format, so we just get the hex number. We want to control the formatting a little more, such as specifying leading zeros. (of course, no programmer is satisfied with what is found in the standard libraries so one must always embrace and extend). Then we may read the NGWSSDK documentation of Format a little closer. It says: ." The format specifications are embedded within a string and are of the form {N:FormatString}, where N is a zero-based number indicating the argument that is to be formatted and FormatString is a format string. If arg0 implements IFormattable, then arg0.Format (FormatString, null) provides the formatting; otherwise arg0.ToString() provides the formatting. If arg0 is null then an empty string is used in its place we see that in classic object oriented programming style, the arguments themselves generate the formatted string, not the Format function. Its not like the code of printf, which is a lexical scanner with dozens of special cases parsing the format string and building the output. Instead, the Format function just looks for the format specifiers, extracts them, and hands them to the objects themselves which then can do whatever they want to.

Considering this, it is evident that all the primitive types have Format methods of their own. And the primitive types implement the IFormattable interface which has single method with the signature:

```
public virtual string Format(string format,IServiceObjectProvider sop);
```

This is the method (of the argument) that gets called by System.String.Format to process the argument. Note that in the quote above,this method is called as argn.Format(FormatString,null). There is no need to go further into IServiceObjectProvider sop (which is pretty hairy), other than to see that it relates to localization. Since Format calls it will that argument == null, the object is expected to use the current locale to format itself.
Thus for hex formatting, one needs an object that when passed to System.String.Format, would format itself in hex.

One might think that one should just extend System.UInt32 and override its Format function and leave the rest alone. But Oops can't do that. UInt32 is a struct and inheritance is not supported for structs. It is generally not advisable. Abstracting the primitive types usually hastens the learning curve for the next person who looks at the code.

So the other approach is to create a type that simply holds the numeric value and exposes a Format method. Here is what we will come up with.

```
s = System.String.Format("{0:#}",new HexFormat(1));
```

This object will take the numeric input when constructed, and when requested by System.String.Format, will generate a formatted string and serve it up.
The important point to remember here is that even though the object is limited in scope, the Format mechanism for our objects will work the same way, so that for any object we have, we can support IFormattable and implement the Format method to produce whatever formatted output makes sense.

REQUIREMENTS

The HexFormat object instance exists only to Format numeric values. Following are the requirements for the object that we started with:

1. format any integral primitive type, both signed and unsigned. This includes byte, ushort, uint, ulong, sbyte, short, int and long.
2. support the '0' and '#' format string characters in the same way they work for decimal integers.
3. default to lower case hex digits. Rather than have some switch for upper or lower, you would find it easier just to implement an extension of the HexFormat that overrides Format and produces an upper case representation.
4. positive hex numbers can be represented with or without leading zeros. The length of the formatted hex number will not exceed its natural max length unless leading. By that I mean, without leading zeros, a byte will be max 2 characters, a short, 4 characters, int, 8 characters and long, 16 characters. Same for unsigned types. If leading zeros are applied in the FormatString, they can extend without arbitrary limit.
5. negative hex numbers will always sign extend at least to their natural length. If leading zeros in excess of the natural length are specified, then there will be sign extension to fill the specified number of characters.
6. a HexFormat, object is immutable once instantiated.

> NOTE: the integer field width component of the format specifier is handled by the System.String.Format method, not by the object argument, so the HexFormat class doesn't have to worry about that. By experimentation, I found that if you have a specifier of "{0,8:###}" the arg.Format method only gets the "###" as it's format string. The 0,8 is handled by System.String.Format. Cool. That makes it easier.

Examples: 1. {0:#} format hex value without leading zeros or spaces (at least 1 digit, up to natural length of the particular type)
2. {0:######} format hex value without leading zeros or spaces (same as 1.)
3. {0:##########} format hex value without leading zeros or spaces (same as 1.)
4. {0:0} format hex value with leading zeros
5. {0:00} format hex value with at least 2 digits. show leading zeros
6. {0:0000} format hex value with at least 4 digits. show leading zeros
7. {0:00000000} format hex value with at least 8 digits. show leading zeros
8. {0:0#######} format hex value with at least 8 digits. show leading zeros (same as 7.)

12.20 Design/Implementation Approach

We have used an iterative design/implementation cycle to develop the HexFormat object. Each cycle is expected to output a compilable and working deliverable, albeit with limited functionality. Here are the following objectives for each cycle:

First Iteration :

1. define the object's public interface and get it to compile and run.
2. do not actually format the data. Use hardcoded output.

Second Iteration :

1. verify that instantiated objects dispatch properly by printing diagnostic strings into the format target (rather than actually formatting some hex)
2. experiment and determine exactly how System.String.Format calls the IFormattable method of the object and what arguments it really passes
3. do not actually format data. Use diagnostic output (the test case can fail, we just want to see the diagnostics)

Third Iteration :

1. implement the hex formatting for all the supported data types
2. execute and verify that the whole thing works
3. implement test examples for all types

First Iteration

Here is the public interface we end up with (see hf1.cs for the actual working implementation). The main features of it are that it has a constructor for each supported type. The constructor will be expected to somehow keep information about what type and value is to be formatted. That info is then used along with the format string argument in the Format method to generate the actual formatted string.

Listing 12.15

```
using System;
using System.IO;
public class HexFormat : System.IFormattable {
    // signed type constructors
    public HexFormat(sbyte i) {}
    public HexFormat(short i) {}
    public HexFormat(int i) {}
    public HexFormat(long i) {}
    // unsigned type constructors
    public HexFormat(byte i) {}
    public HexFormat(ushort i) {}
    public HexFormat(uint i) {}
    public HexFormat(ulong i) {}
```

```
    // IFormattable implementation
    // hard coded for now to support the test case
    public virtual  string Format(string f,IServiceObjectProvider
sop) {return "1";}
    public override string ToString() {return "1";}
    // internal class test fixture
    public static void Main(string[] args) {
        // example usage of HexFormat object
        String h = System.String.Format("{0:#}",new
HexFormat(1));
        if (h == "1") {
            System.Console.Write("ok  : " + h);
        }
        else {
            System.Console.Write("format failed  : " + h);
        }
    }
}
```

Notes:

1. once constructed, the value contained in the HexFormat object is immutable.
2. the work for formatting can take place in some combination of the constructor and the Format method. At this point, I don t care how its done.
3. the goal for the first cycle is met
4. the interface satisfies requirement #1 (which types to support)
5. the interface satisfies requirement #6 (there are no mutators for the object)

Second Iteration

The public interface is not changed. The approach is that the constructor calls a private method 'convert', with a separate 'convert' method for signed and unsigned types, just in case they end up needing to be handled differently. If it turns out that a single 'convert' is all that is needed, then cool. The 'convert' method in this case outputs diagnostics that show the constructor knows the type and the natural size of the type in formatted digits. The Format function then prints a diagnostic output that includes the data the constructor generated, plus the format string from System.String.Format.

see file hf2.cs for the implementation of the second phase

Third Iteration

The public interface is not changed. Now we are implementing the real thing for the 32 bit 'int' type. We had to decide on an actual conversion algorithm.

Getting the value converted to hex is easy. The class System.Convert has methods to convert all the signed types plus 'byte' to any base. So those are used to convert to base 16. This is done in the constructor because that is where we know the exact type of the value.

Also in the constructor, the character needed for leading padding is figured out and saved. For signed types, it can be 'f' or '0'. For unsigned types, its always '0';

The Format method is responsible for matching up the contents of the stack with the format string to produce the proper output string.

We notice the following rules that we could use:

1. since a hex number has no fractional digits, any length format string of #'s is the same as a single #
2. once a 0 is encountered in the format string, the rest of the characters in the format string are treated as 0, since you can't skip 0 digits in the middle of a number.
3. #'s before 0's in the format string can be ignored

So our algorithm for formatting becomes:

```
Iterate thru the format string from front to back until a '0',
     invalid format character or end of string is found.
If the format has anything but # or 0, it is invalid, throw an exception
If a zero is in the format string,
    then the position of the zero determines the minimum length
of the string.
        the formula is LengthOfFormatString -
ZeroBasedPositionOfFirstZero = min output length
    else if there is no zero in the format string
        then the minimum length is 1
for i=0 to MinOutputStringLength - NumberOfDigitsInTheString
    output the leading pad character
append the actual digit string from the constructor
done.
```

This program can be used to create a colour-coded, HTML version of your source code, in just about any language—JScript, C#, C++, Visual Basic, and more!
This new version has some great features:

No support files required to use all the default options! Improved support for colouring numbers and compiler directives Fully extensible INI and command-line driven architecture Dynamic replacement of strings in the output file Pre-built style sheets based on the Microsoft and Borland colour schemes Still runs under WSH with JScript 5.5 or the PDC release of JScript.NET To see how the formatter formated its own source code, click here.

Extensible architecture Anything you add in the INI file or on the command-line gets added to the global g—objOptions object, so you can use it in a dynamic replacement, as specified in the file replace.ini. You can change the name of the options file with the /ini command-line argument, and in the options file you can change the name of any of the other INI files.

Check out the default replace.ini and header.txt files to see the sorts of things that you can do.

Sample command-lines:

Listing 12.16

```
formatCodeAsHtml.exe formatCodeAsHtml.js (result) formatCodeAsHtml.exe
formatCodeAsHtml.js  /stylesheet:microsoft.css format.microsoft.html
(result) formatCodeAsHtml.exe /ini:vb_options.ini /stylesheet:blue.css
"/language:Visual Basic" simple.vb (result) formatCodeAsHtml.exe /
ini:c_options.ini /stylesheet:microsoft.css simple.cpp (result)
```

A Component in C# Here is how our simple string component looks in C#:

Listing 12.17:Component in C# (CompCS.cs)

```
using System;
namespace CompCS {
   public class StringComponent {
      private string[] StringsSet;
      public StringComponent() {
         StringsSet = new string[] {
            "C# String 0",
            "C# String 1",
            "C# String 2",
            "C# String 3"
         };
      }
     public string GetString(int index) {
        if ((index < 0) || (index >=
           StringsSet.Length)) {
           throw new IndexOutOfRangeException();
         }
         return StringsSet[index];
      }
      public int Count {
         get { return StringsSet.Length; }
      }
   }
}
```

As mentioned above, we use the namespace statement to create a new namespace to encapsulate the classes we will be creating:

namespace CompCS {_} This namespace may be nested and may be split between multiple files: A single source code file may also contain multiple non-nested namespaces. A containing namespace is required, since all C# code must be contained in a class.

public class StringComponent {_} This statement means that instances of StringComponent will now be created by the runtime and managed in the Garbage Collected heap. The class constructor which executes each time a new instance of the class is created has the same name as the class and doesn't have a return type.

public StringComponent() {_} Since C# uses only managed types, we don't have to do anything special—as we had to do with Managed Extensions—when declaring variables.

Here is the GetString method, which takes an integer and returns a string:

```
public string GetString(int index) {
   _
    return StringsSet[index];
}
```

Note the throw statement in the GetString method:
throw new IndexOutOfRangeException(); This statement creates and throws a runtime-based exception handling object of type IndexOutOfRangeException.

Another way of returning strings would be to use an indexer instead of the GetString method:

Listing 12.18

```
public string this[int index] {
   if ((index < 0) || (index >= Count)) {
      throw new IndexOutOfRangeException();
   }
   return StringsSet[index];
}
```

An indexer would allow the client to use a much more natural syntax of:
StringSetVar[index] VB, however, doesn't support indexers so, in the interest of keeping the component code the space for each of the three languages, we have chosen to use a distinct GetString method.

Finally, we create the read-only property Length:

```
public int Count {
   get { return StringsSet.Length; }
}
```

The compile process for a C# component is only a little more complicated than for a stand-alone program:
csc /out:..\Bin\CompCS.dll /target:library CompCS.cs As with the C++ component, we use the /out switch to put the compiled component in the "..\Bin" relative subdirectory for convenience. Likewise, we need the /target:library switch to actually create a DLL rather than an executable with a .dll file extension.

A Client in C#: Here is how our client looks in C#:

Listing 12.19: Client in C# (ClientCS.cs)

```
using System;
using CompVC; using CompCS; using CompVB;
// This "class" exists to house application's entry-point class MainApp
{
  // Static method "Main" is application's entry point
  public static void Main() {
    // Iterate over component's strings dump to console
    CompCS.StringComponent myCSStringComp = new
        CompCS.StringComponent();
    Console.WriteLine("Strings from C# StringComponent");
    for (int index = 0; index < myCSStringComp.Count;
        index++) {
      Console.WriteLine(myCSStringComp.GetString(index));
    }
    // Iterate over component's strings dump to console
    CompVC.StringComponent myVCStringComp = new
        CompVC.StringComponent();
    Console.WriteLine("\nStrings from VC
       StringComponent");
      for (int index = 0; index < myVCStringComp.Count;
         index++) {
       Console.WriteLine(myVCStringComp.GetString(index));
      }
    // Iterate over component's strings dump to console
    CompVB.StringComponent myVBStringComp = new
        CompVB.StringComponent();
    Console.WriteLine("\nStrings from VB
        StringComponent");
    for (int index = 0; index < myVBStringComp.Count;
        index++) {
      Console.WriteLine(myVBStringComp.GetString(index));
    }
  }
}
```

Unlike the Managed Extensions example, we don't have to import the libraries at this point: Instead, we can specify them in the compile process. The advantage of specifying a library with the using statement is that, by incorporating the namespace into the program, we can then reference types in the library without fully qualifying the type name. Since our particular example has the same type name (StringComponent) in each of the components, we still have use the fully qualified name to remove any ambiguity when referring to the method (GetString) and property (Count), which are common to each component. C# also provides a mechanism called aliasing to address this problem: If you changed the using statements to:

using VCStringComp = CompVC.StringComponent;

using CSStringComp CompCS.StringComponent;

using VBStringComp = CompVB.StringComponent;

then you would not have to fully qualify the names.

The client code is virtually identical to the C++ example except for the scope resolution operators. Again, the code that calls the three string components is also the same except for specifying which library to use. As with the Managed Extensions example, the first statement in each of the three sections declares a new local variable of a type StringComponent, initializes it, and calls its constructor::

```
CompCS.StringComponent myCSStringComp = new
   CompCS.StringComponent();
```

After writing out a string to the console to say we're entering this part of the program, the client then iterates over the members—up to the value of the Count property—of appropriate string component:

```
for (int index = 0; index < myVCStringComp.Count;
    index++) {
   Console.WriteLine(myVCStringComp.GetString(index));
}
```

That's all that's required, and everything's repeated for the other two language components. Building our new C# client is straightforward. Now that we are using components in our own relative "..\Bin" subdirectory, we need to explicitly include them using the /reference compilation switch:

```
csc /reference:..\Bin\CompCS.dll;..\Bin\CompVB.dll;
   ..\Bin\CompVC.dll /out:..\Bin\ClientCS.exe
   ClientCS.cs
```

Also, since we're building a client program rather than a component that might be called from other programs, it is not necessary to use the /cls+ switch.

Other than that, the process is the same as with the previous C# examples. Running the resulting program yields:

Listing 12.20

```
C:\_\FrameworkSDK\Samples\IntroDev\CompTest\Bin>clientcs Strings from
C# StringComponent C# String 0 C# String 1 C# String 2 C# String 3
Strings from VC StringComponent VC String 0 VC String 1 VC String 2 VC
String 3
Strings from VB StringComponent VB String 0 VB String 1 VB String 2 VB
String 3
```

13

ATTRIBUTES

13.1 Attributes

C# is a procedural language.Like all procedural languages it does have some declarative elements. For example, the accessibility of a method in a class is specified by decorating it public, protected, internal, protected internal, or private. By supporting attributes, C# generalizes this capability, so that programmers can invent new kinds of declarative information, specify them for various program entities, and retrieve them at run-time. Programs specify this additional declarative information by defining and using attributes.

In most programming languages, some information is expressed through a declaration, and others through a code. For example, in the class member declaration:

```
public int Test;
```

the compiler and runtime will reserve space for an integer variable and set its protection so that it is visible everywhere. This is an example of declarative information; it's appreciable for the economy of expression and because the compiler handles the details for us.

Typically, the types of declarative information are predefined by the language designer and can't be extended by the users of the language. A user who wants to associate a specific database field with a field of a class, for example, must invent a way of expressing that relationship in the language, a way of storing the relationship, and a way of accessing the information at runtime. In a language like C++, a macro might be defined that stores the information in a field that is part of the object. Such schemes work, but they are error-prone and not generalized. They are also unweildy.

The .NET Runtime supports attributes, which are merely annotations that are placed on elements of source code, such as classes, members, parameters, etc. Attributes can be used to change the behavior of the runtime, provide transaction information about an object, or convey organizational information to a designer. The attribute information is stored with the metadata of the element and can be easily retrieved at runtime through a process known as reflection.

C# uses a conditional attribute to control when member functions are called. A use for the conditional attribute would look like this:

Listing 13.1

```
using System.Diagnostics;
class Test {
    [Conditional("DEBUG")]
    public void Validate()
    {
    }
}
```

Most programmers prefer predefined attributes to writing attribute classes.

Generally the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the method-modifiers public, protected, internal, and private.

The run-time environment often requires the programmer to improvise declarative information for various program entities and to retrieve attribute information. For instance, a framework might require defining a HelpAttribute attribute that can be placed on program elements such as classes and methods to provide a mapping from program elements to documentation for them.

New kinds of declarative information are defined through the declaration of attribute classes, which may have positional and named parameters. Declarative information is specified in a C# program using attributes, and can be retrieved at run-time as attribute instances.

A mechanism for defining declarative tags, called attributes, which you can place on certain entities in your source code to specify additional information is also available. The information contained in these attributes can be retrieved at run time through reflection. You can use predefined attributes or you can define your own custom attributes.

Attributes can be placed on most declarations (though a specific attribute might restrict the types of declarations on which it is valid). Syntactically, an attribute is specified by placing the name of the attribute, enclosed in square brackets, in front of the declaration of the entity to which it applies. For example, a class with the attribute DllImport is declared as follows:

```
[DllImport] public class MyDllimportClass { ... }
```

Many attributes have parameters, which can be either positional (unnamed) or named. Positional parameters must be specified in a certain order and cannot be omitted; whereas named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, the following three attributes are equivalent:

```
[DllImport("user32.dll", SetLastError=false,   ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false,   SetLastError=false)]
[DllImport("user32.dll")]
```

Here,the first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted (refer to the individual attribute's documentation for information on default parameter values).

More than one attribute can be placed on a declaration, either separately or within the same set of brackets:

```
bool AMethod([In][Out]ref double x);
bool AMethod([Out][In]ref double x);
bool AMethod([In,Out]ref double x);
```

Some attributes can be specified more than once for a given entity, that is, they are multi-case attributes. An example of this is, 'Conditional', which is as follows:

```
[Conditional("DEBUG"), Conditional("TEST1")] void TraceMethod()  {...}
```

Most attributes are attached to specific language elements, say, classes or methods; however, some are global—they apply to an entire assembly or module. Assembly-level attributes are specified as follows:

[assembly: attribute-list] Module-level attributes are specified as follows:

```
[module: attribute-list]
```

where attribute-list is a list of applicable global attributes.

Global attributes appear in the source code after any top-level using directives and before any type or namespace declaration. They can also appear in multiple source files in a single compilation.

Note that assembly-level attributes can be specified only if an assembly manifest is being created.The following .NET attributes let you modify the information in an assembly:

```
System.Reflection.AssemblyConfigurationAttribute
System.Runtime.CompilerServices.AssemblyCultureAttribute
System.Reflection.AssemblyDescriptionAttribute
System.Runtime.CompilerServices.AssemblyKeyFileAttribute
System.Runtime.CompilerServices.AssemblyKeyNameAttribute
System.Runtime.CompilerServices.AssemblyDelaySignAttribute
System.Reflection.AssemblyTitleAttribute
System.Runtime.CompilerServices.AssemblyVersionAttribute
```

If you do not want an assembly created for a compilation, use the compiler option /target:module. For example:

```
[assembly: System.CLSCompliant(true)]
```

13.2 Creating Custom Attributes

Specific custom attributes can be created by defining an attribute class, a class that derives directly or indirectly from System.Attribute (which makes identifying attribute definitions in metadata fast and easy). For instance, if you want to tag classes and structs with the name of the programmer who wrote the class or struct. You might define a custom Author attribute class:

Listing 13.2

```
using System;
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct)]
public class Author : Attribute {
   public Author(string name) { this.name = name; version = 1.0;
}
```

```
    public double version;
    string name;
}
```

The class name is the attribute's name: Author. It is derived from System.Attribute, so it is a custom attribute class. The constructor's parameters are the custom attribute's positional parameters (in this case, name), and any public read-write fields or properties are named parameters (in this case, version is the only named parameter). Note that the use of the AttributeUsage attribute to make the Author attribute is valid only on class and struct declarations.You can use this new attribute as follows:
[Author("Hemlata", version=1.1)] class SomeClass{...}

AttributeUsage has a named parameter, AllowMultiple, with which you can make a custom attribute single-use or multiuse.
Listing 13.3

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct,
    AllowMultiple=true)]
// multiuse attribute
public class Author : Attribute {
    ...
}
```

Note: To prevent namespace collisions, all attribute names implicitly end with 'Attribute'; in the preceding example, Author and AuthorAttribute could be used interchangeably. The same applies to AttributeUsage, which is defined in the .NET Framework as System.AttributeUsageAttribute.

13.3 Retrieving Attribute Information

The fact that you can define custom attributes and place them in your source code would be of little use unless there is some way of retrieving that information and acting on it. C# has a reflection system for this. The key method is GetCustomAttributes, which returns an array of objects that are the run-time equivalents of the source code attributes.

An attribute specification such as:

```
[Author("Hemlata", version=1.1)] class AClass {...}
```

is conceptually equivalent to this:

```
Author anonymousAuthorObject = new Author("Hemlata");
anonymousAuthorObject.version = 1.1;
```

However, the code is not executed until AClass is queried for attributes. Calling GetCustomAttributes on AClass causes an Author object to be constructed and initialized as above. If the class has other attributes, other attribute objects are constructed similarly. GetCustomAttributes then returns the Author object and the other attribute objects in an array. You can then iterate over this array, determine what attributes were applied based on the type of each array element, and extract information from the attribute objects.

Example

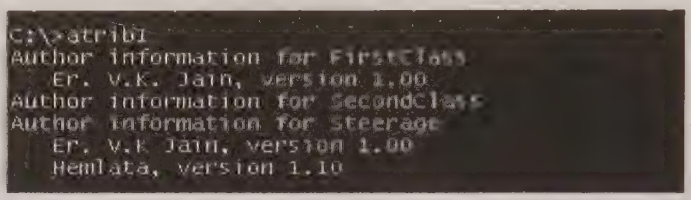
Here is a complete example. A custom attribute is defined, applied to several entities, and retrieved via reflection.

Listing 13.4: Example on Attributes

```
//atrib1.cs
using                                                    System;
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct,
   AllowMultiple=true)]
public class Author : Attribute {
   public Author(string name)
   {
      this.name = name; version = 1.0;
   }
   public double version;
   string name;
   public string GetName()
   {
      return name;
   }
}
[Author("Er. V.K. Jain")] class FirstClass {
   /*...*/
}
class SecondClass  // no Author attribute {
   /*...*/
}
[Author("Er. V.K Jain"), Author("Hemlata", version=1.1)] class Steer-
age {
   /*...*/
}
class AuthorInfo {
   public static void Main()
   {
      PrintAuthorInfo(typeof(FirstClass));
      PrintAuthorInfo(typeof(SecondClass));
      PrintAuthorInfo(typeof(Steerage));
   }
   public static void PrintAuthorInfo(Type t)
   {
      Console.WriteLine("Author information for {0}", t);
      Attribute[] attrs = Attribute.GetCustomAttributes(t);
      foreach(Attribute attr in attrs)
      {
         if (attr is Author)
```

```
            {
                Author a = (Author)attr;
                Console.WriteLine("   {0}, version {1:f}",
a.GetName(), a.version);
            }
        }
    }
}
```

The output of this program will get displayed as:



13.4 Using Attributes

Placing the AttributeUsage attribute on an attribute class controls where the attribute can be used. The possible values for the attribute are listed in the AttributeTargets enumerator and are as follows:

Value	Meaning
Assembly	The program assembly
Module	The current program file
Class	A class
Struct	A struct
Enum	An enumerator
Constructor	A constructor
Method	A method (member function)
Property	A property
Field	A field
Event	An event
Interface	An interface
Parameter	A method parameter
Return	The method return value
Delegate	A delegate
All	Anywhere ClassMembers Class, Struct, Enum, Constructor, Method, Property, Field, Event, Delegate, Interface

As part of the AttributeUsage attribute, one of these can be specified or a list of them can be ORed together.

The AttributeUsage attribute is also used to specify whether an attribute is single-use or multi-use. This is done with the named parameter AllowMultiple. Such an attribute would look like this:

```
[AttributeUsage( AttributeTargets.Method |  AttributeTargets.Event,
                 AllowMultiple = true)]
```

Suppose that for a project that a group was doing, it was important to keep track of the code reviews that had been performed on the classes so that it could be determined when code reviews were finished. The code review information could be stored in a database, which would allow easy queries about status, or it could be stored in comments, which would make it easy to look at the code and the information at the same time.

Or an attribute could be used, which would enable both kinds of access.

For this, an attribute class is needed. An attribute class defines the name of an attribute, how it can be created, and the information that will be stored. The gritty details of defining attribute classes will be taken up in the section entitled 'An Attribute of Your Own.'

The attribute class will appear as in the following Listing:

Listing 13.5

```
using System;
[AttributeUsage(AttributeTargets.Class)]
 public class CodeReviewAttribute: System.Attribute
{
    public CodeReviewAttribute(string reviewer, string date)
    {
        this.reviewer = reviewer;
        this.date = date;
    }
    public string Comment
    {
        get
        {
            return(comment);
        }
        set
        {
            comment = value;
        }
    }
    public string Date
    {
        get
        {
```

```
            return(date);
        }
    }
    public string Reviewer
    {
        get
        {
            return(reviewer);
        }
    }
    string reviewer;
    string date;
    string comment;
}
[CodeReview("Eric", "01-12-2000", Comment="Bitchin' Code")] class Com-
plex { }
```

The AttributeUsage attribute before the class implies that this attribute can only be placed on classes. When an attribute is used on a program element, the compiler checks to see whether the use of that attribute on that program element is allowed.

The naming convention for attributes is to append Attribute to the end of the class name. This makes it easier to tell which classes are attribute classes and which classes are normal classes. All attributes must derive from System.Attribute.

The class defines a single constructor that takes a reviewer and a date as parameters, and it also has the public string Comment.

When the compiler comes to the attribute usage on class Complex, it first searches for a class derived from Attribute named CodeReview. It doesn't find one. So it will try for a class named CodeReviewAttribute, which it will find.

Next, it checks whether the attribute is allowed on a class.

Then it goes on to check whether there is a constructor that matches the parameters we've specified in the attribute use. If it finds one, an instance of the object is created-the constructor is called with the specified values.

If there are named parameters, it matches the name of the parameter with a field or property in the attribute class, and sets the field or property to the specified value.

After this the current state of the attribute class is saved to the metadata for the program element for which it was specified.

This is the logical sequence. Please see the 'Attribute Pickling' sidebar for a description of how it is implemented.

13.5 Attribute Usage

Describes how a custom attribute class can be used.

```
[AttributeUsage(
 validon,
 AllowMultiple=allowmultiple,
 Inherited=inherited
)]
```

Parameters validon specifies the language elements on which the attribute can be placed; a combination of AttributeTargets values. Default value is AttributeTargets.All.

```
allowmultiple (optional
```

A bool; if true, the attribute is multiuse. Default is false (single-use). inherited (optional)
A bool; if true, the attribute is inherited by derived classes. Default is false (not inherited). Applies To Class declarations.

Remarks The AttributeUsage attribute is a single-use attribute. AttributeUsage is an alias for System.AttributeUsageAttribute.

13.6 Attribute Pickling

For the compiler to actually create the attribute object, the .NET Runtime environment would have to be running, so every compilation would have to start up the environment, and every compiler would have to run as a managed executable.

Additionally, the object creation isn't really required, since we're just going to store the information away.

The compiler therefore validates that it could create the object, call the constructor, and set the values for the named parameters. The attribute parameters are then pickled into a chunk of binary information, which is tucked away with the metadata of the object.

13.7 An Attribute of Your Own

To define attribute classes and reflect on them at runtime, a few more issues are to be considered. This section will discuss some considerations pertinent to designing an attribute.

There are two major aspects with regard to writing an attribute. The first is the program elements that the attribute may be applied to, and the second is the information that will be stored by the attribute.

13.8 Attribute Parameters

The information the attribute will store should be divided into two groups: the information that is required for every use, and the optional items.

The information that is required for every use should be obtained through the constructor for the attribute class. This forces the user to specify all the parameters when they use the attribute.

Optional items should be implemented as named parameters, for this allows the user to specify appropriate optional items.

If an attribute can be created in different ways for different information required, separate constructors too can be declared for each usage. But separate constructors should not be used as alternatives to optional items.

13.9 Attribute Parameter Types

The attribute pickling format supports only a subset of all the .NET Runtime types, and therefore, only some types can be used as attribute parameters. Thefollowing types are allowed.

- bool, byte, char, double, float, int, long, short, string
- object
- System.Type
- An enum that has public accessibility (not nested inside something non-public)
- A one-dimensional array of one of the above types

13.10 Reflecting on Attributes

Once attributes are defined on some code, it is desirable to find the attribute values. This is done through reflection.

The following code shows an attribute class, the application of the attribute to a class, and the reflection on the class to retrieve the attribute.

Listing 13.6

```
using System;
using System.Reflection;
[AttributeUsage(AttributeTargets.Class)]
public class CodeReviewAttribute: System.Attribute
{
    public CodeReviewAttribute(string reviewer, string date)
    {
        this.reviewer = reviewer;
        this.date = date;
    }
    public string Comment
    {
        get
        {
            return(comment);
        }
        set
        {
            comment = value;
        }
    }
    public string Date
    {
        get
```

```
        {
            return(date);
        }
    }
    public string Reviewer
    {
        get
        {
            return(reviewer);
        }
    }
    string reviewer;
    string date;
    string comment;
} [CodeReview("V.K. Jain", "24-03-2001", Comment="Excellent")] class
Complex { } class Test {
    public static void Main()
    {
        System.Reflection.MemberInfo info;
        info = typeof(Complex);
        object[] atts;
        atts =
info.GetCustomAttributes(typeof(CodeReviewAttribute));
        if (atts.GetLength(0) != 0)
        {
            CodeReviewAttribute att = (CodeReviewAttribute) atts[0];
            Console.WriteLine("Reviewer: {0}", att.Reviewer);
            Console.WriteLine("Date: {0}", att.Date);
            Console.WriteLine("Comment: {0}", att.Comment);
        }
    }
}
```

The Main() function first gets the type object associated with the type Complex. It then loads all the attributes that are of the CodeReviewAttribute type. If the array of attributes has any entries, the first element is cast to a CodeReviewAttribute, and then the value is printed out. There can only be one entry in the array because CodeReviewAttribute is single-use.
This example produces the following output:

```
Reviewer: V.K. Jain Date: 24-03-2001 Comment: Excellent
```

GetCustomAttribute() can also be called without a type, to get all the custom attributes on an object.

Note: The "CustomAttributes" in the preceding example refers to attributes that are stored in a general attribute part of the metadata for an object. Some .NET Runtime attributes are not stored as custom attributes on the object, but are converted to metadata bits on the object. Runtime reflection does not support viewing these attributes through reflection. This restriction may be addressed in future versions of the runtime.

For instance, a framework might define a HelpAttribute attribute that can be placed on program elements such as classes and methods, so as to enable developers to provide a mapping from program elements to documentation for them. See the following example:
Listing 13.7

```
using System;
[AttributeUsage(AttributeTargets.All)] public class HelpAttribute: At-
tribute {
     public HelpAttribute(string url) {
          this.url = url;
     }
     public string Topic = null;
     private string url;
     public string Url {
          get { return url; }
     }
}
```

Here the attribute class named HelpAttribute, or Help for short, that has one positional parameter (string url) and one named argument (string Topic). Positional parameters are defined by the formal parameters for public constructors of the attribute class, and named parameters are defined by public read-write properties of the attribute class.
The example

```
[Help("http://www.mycompany.com/_/Class1.htm")] public class Class1 {
     [Help("http://www.mycompany.com/_/Class1.htm", Topic = "F")]
     public void F() {}
}
```

illustrates various uses of the attribute. Attribute information for a given program element can be retrieved at run-time by using reflection support.
Listing 13.8

```
using System; class Test {
     static void Main() {
          Type type = typeof(Class1);
          object[] arr =
type.GetCustomAttributes(typeof(HelpAttribute));
          if (arr.Length == 0)
    Console.WriteLine("Class1 has no Help attribute.");
          else {
        HelpAttribute ha = (HelpAttribute) arr[0];
    Console.WriteLine("Url = {0}, Topic = {1}", ha.Url, ha.Topic);
          }
     }
}
```

The listing checks whether Class1 has a Help attribute, and writes out the associated Topic and Url values if the attribute is present.

13.11 Interfaces

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. Interfaces can contain methods, properties, indexers, and events as members. An interface is a declaration of a collection of class members. They do not contain implementation; instead they indicate that a class that implements the interface must provide those members.

Listing 13.9

```
interface IExample {
     string this[int index] { get; set; }
     event EventHandler E;
     void F(int value);
     string P { get; set; }
} public delegate void EventHandler(object sender, EventArgs e);
shows an interface that contains an indexer, an event E, a method F,
and a property P.
Interfaces may employ multiple inheritance. In the example
interface IControl {
     void Paint();
} interface ITextBox: IControl {
     void SetText(string text);
} interface IListBox: IControl {
     void SetItems(string[] items);
} interface IComboBox: ITextBox, IListBox {} the interface IComboBox
inherits from both ITextBox and IListBox. Classes and structs can
implement multiple interfaces.
In the example
interface IDataBound {
     void Bind(Binder b);
} public class EditBox: Control, IControl, IDataBound {
     public void Paint() {_}
     public void Bind(Binder b) {_}
}
```

the class EditBox derives from the class Control and implements both IControl and IDataBound.

In the previous example, the Paint method from the IControl interface and the Bind method from IDataBound interface are implemented using public members on the EditBox class. In an alternative way of implemening these methods, the implementing class need not have these members as public. Interface members can be implemented using a qualified name. For example, the EditBox class could instead be implemented by providing IControl.Paint and IDataBound.Bind methods.

Interface members implemented in this way are called explicit interface members because each member explicitly designates the interface member being implemented. Explicit interface members can only be called via the interface.

The NamedClass class can be used like this:

```
NamedClass c = new NamedClass(); c.PrintName(); INamed ic = c;
ic.PrintName();
```

The output from PrintName() is the same in both the cases. You can prevent this by ensuring that the interface members are accessed only through an interface. For this, you may qualify the method name with the interface name.

A class can implement more than one interface, so if you have another class that implements INamed and IViewable the code can just ask for the behavior determined by one of these interfaces, for example INamed:

Listing 13.14

```
public class NVOne : INamed, IViewable { /* implement INamed and
IViewable members*/ } public class NVTwo : INamed, IViewable { /*
implement INamed and IViewable members*/ } public class A {
   private void f(INamed n)
   {
      n.PrintName();
   }
   public void g()
   {
      NVOne n1 = new NVOne();
      f(n1);
      NVTwo n2 = new NVTwo();
      f(n2);
   }
}
```

Interfaces allow you to select the behavior that you want, in this code the method A.g() creates two objects, one each of two different types. These two objects are passed to the method A.f() that uses just the INamed behavior of each of these objects. Interface programming like this is very powerful, indeed, it is the basis of COM. Classes can have a base class in addition to indicating that it implements interfaces. So, if NamedBase has the code for the INamed interface members as follows:

Listing 13.15

```
public class NamedBase
{
   protected String name = "empty";
   public String Name
   {
      get{return name;}
   }
   public void PrintName()
   {
      Console.WriteLine("Name is {0}", name);
   }
}
```

You can use this with classes that implement the INamed interface, which will inherit the NamedBase implementation:

Listing 13.16

```
public class One : NamedBase, INamed
{
   public One()
   {
      name = "One";
   }
} public class Two : NamedBase, INamed {
   public Two()
   {
      name = "Two";
   }
}
```

These classes could be used like this:

Listing 13.17

```
INamed[] namedObjects = {new One(), new Two()};
foreach (INamed o in namedObjects)
   o.PrintName();
```

This creates an array with instances of the two object types. The foreach construct is used to access all the members in the array, and since the members are INamed, the code can call PrintName() irrespective of the actual type of the object.

13.12 Delegates

A delegate in C# is very similar to a function pointer in C or C++. A delegate can be used to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked. Unlike function pointers in C or C++, delegates are object-oriented, type-safe, and secure.

A delegate declaration defines a type that encapsulates a method with a particular set of arguments and return type. With static methods, a delegate object encapsulates the method to be called. With instance methods, a delegate object encapsulates both an instance and a method on the instance. If you have a delegate object and an appropriate set of arguments, you can invoke the delegate with the arguments.

An interesting and useful property of a delegate is that it is not sensitive to the class of the object that it references. Any object will do; as long as the argument type of the method and the return type match with the delegate. This makes delegates perfectly suited for 'anonymous' invocation.

13.13 DelegateTypes

A delegate is a data structure that refers to a static method or to an object instance and an instance method of that object.

The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only reference static functions, a delegate can reference both static and instance methods. In the latter case, the delegate stores both; a reference to the method's entry point, and a reference to the object instance for which to invoke the method.

Unlike C++ function pointers, delegates are fully object oriented. Besides, delegates encapsulate both an object instance and a method.

A delegate declaration defines a class that extends the class System.Delegate. A delegate instance encapsulates a method—a callable entity. For instance, methods—a callable entity—consist of an instance and a method on the instance. For static methods, a callable entity consists of just a method. If there is a delegate instance and an appropriate set of arguments, one can invoke the delegate with the arguments.

An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's signature matches the delegate's. This makes delegates perfectly suited for 'anonymous' invocation.

Delegates enable scenarios that C++ and some other languages have addressed with function pointers. Unlike function pointers, delegates are object-oriented, type-safe, and secure. Delegates are reference types that derive from a common base class: System.Delegate. A delegate instance encapsulates a method-a callable entity. For instance, methods, a callable entity consists of an instance and a method on the instance. For static methods, a callable entity consists of a class and a static method on the class.

An interesting and useful property of a delegate is that it does not know or care about the type of the object that it references. Any object will do; all that matters is that the method's signature matches the delegate's. This makes delegates perfectly suited for 'anonymous' invocation. This is a powerful capability.

There are three steps in defining and using delegates:

- Declaration
- Instantiation
- Invocation.

Delegates are declared using delegate declaration syntax.

Listing in Example 13.1 shows how to declare, instantiate, and call a delegate.

Example

The following example illustrates declaring, instantiating, and using a delegate. The BookDB class encapsulates a bookstore database that maintains a database of books. It exposes a method ProcessPaperbackBooks, which finds all paperback books in the database and calls a delegate for each one. The delegate type used is called ProcessBookDelegate. The Test class uses this class to print out the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is done on the paperback books after it finds them.

Listing 13.20

```
// book1.cs
using System;
//Designing classes for handling a bookstore.
namespace Bookstore {
   using System.Collections;
   // Describes details in  book list.
public struct Book {
         public string Title;
         // Title
         public string Author;
         // Author
         public string Editors;
         // Editors
         public decimal Price;
         // Price  public
         bool Paperback;
         // paperback?
         public Book(string title, string author, decimal price, bool
         paperBack)
         {
               Title = title;
               Editors= editors;
               Author = author;
               Price = price;
               Paperback = paperBack;
         }
   }
   // declaring a delegate type for processing a book
   public delegate void ProcessBookDelegate(Book book);
   // Maintains a book database.
   public class BookDB
   {
         ArrayList list = new ArrayList();
         // list of all books in the  database
         // Add a book to the database.
         public void AddBook(string title, string author, editors,
         decimal price,
         bool paperBack)
```

```
                {
                        list.Add(new Book(title, author, editors, price,
                                paperBack));
                }
// Call a passed in delegate on each paperback book to process it
                public void ProcessPaperbackBooks(ProcessBookDelegate
                        processBook)
                {
                        foreach (Book b in list)
                        {
                                if (b.Paperback) processBook(b);
                        }
                }
        }
}
// Using the Bookstore classes
namespace BookTestClient {
        using Bookstore;
        // totalling and average prices of books
        class PriceTotaller
        {
                int countBooks = 0;
                decimal priceBooks = 0.0m;
                internal void AddBookToTotal(Book book)
                {
                        countBooks += 1; priceBooks += book.Price;
                }
                internal decimal AveragePrice()
                {
                        return priceBooks / countBooks;
                }
        }
        // testing the book database
        class Test
        { // Printing title of the book
                static void PrintTitle(Book b)
                {
                        Console.WriteLine(" {0}", b.Title);
                }
                // Execution starts here
                static void Main() {
                        BookDB bookDB = new BookDB();
                        // Initializing the database with some books
                        AddBooks(bookDB);
```

```
                // Printing all the titles of paperbacks
                Console.WriteLine("Paperback Book Titles:");
                bookDB.ProcessPaperbackBooks(new
                      ProcessBookDelegate(PrintTitle));
//Get average price of a paperback by using a PriceTotaller object
PriceTotaller totaller = new PriceTotaller();
   bookDB.ProcessPaperbackBooks(new //line97
   ProcessBookDelegate(totaller.AddBookToTotal));
Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
totaller.AveragePrice());
                }
// Initializing the book database with some test books
                static void AddBooks(BookDB bookDB) {
                      bookDB.AddBook("Java Server Pages",
                      " Asim and Rajan" ,"V.K. Jain", 30.00m, true);
                bookDB.AddBook("Programming WAP,Wap servlets",
                " Asim and Rajan" "V.K. Jain",
                      99.00m, true);
                            bookDB.AddBook("System Analysis and Design",
                            " Asim and Rajan" "V.K. Jain", 99.00m,
                      false);
                      bookDB.AddBook("C# Programming"," Asim and Rajan"
"V. K. Jain", 20.00m, true);
          }
   }
}
```

The output of this program will be displayed as:

```
C:\>book1
Paperback Book Titles:
 Java Server Pages
 Programming WAP,Wap servlets
 C# Programming
Average Paperback Book Price: $49.67
```

Declaring a delegate: Each delegate type describes the number and types of the arguments, and the type of the return value, of methods that it can encapsulate. Whenever a new set of argument types or return value type is needed, a new delegate type must be declared.

Instantiating a delegate: Once a delegate type has been declared, a delegate object must be created and associated with a particular method. Like all other objects, a new delegate object is created with a new expression. When creating a delegate, however, the argument passed to the new expression is special—it is written like a method call, but without the arguments to the method. In both cases, this new delegate object is immediately passed to the ProcessPaperbackBooks method.

Note that once a delegate is created, the method it is associated with never changes— delegate objects are immutable.

Calling a delegate: Once a delegate object is created, it is typically passed to other code that will call the delegate. A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate.

Listing in example 13.2 shows how to combine two delegates.

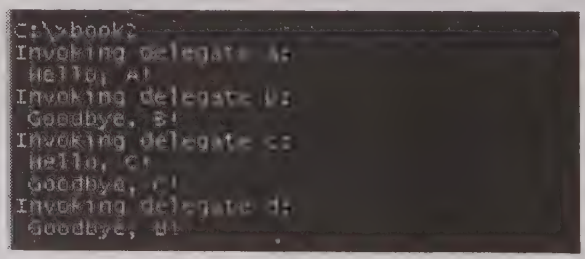
Example

This example demonstrates composing delegates. A useful property of delegate objects is that they can be composed using the "+" operator. A composed delegate calls, in order, the two delegates it was composed from. Only delegates of the same type can be composed, and the delegate type must have a void return value. The "-" operator can be used to remove a component delegate from a composed delegate.

Listing 13.21

```
// book2.cs
using System;
delegate void MyDelegate(string s);
class MyClass {
public static void Alright(string s) {
Console.WriteLine(" Alright, {0}!", s); }
public static void OK(string s) {
Console.WriteLine(" OK, {0}!", s); }
public static void Main() {
MyDelegate alpha, beta, gamma, delta;
alpha = new MyDelegate(Alright);
//line21
beta = new MyDelegate(OK); gamma = alpha + beta;
// Compose two delegates to make another
delta = gamma - alpha;
// Remove a from the composed delegate
Console.WriteLine("starting delegate alpha:");
alpha("Alpha");
Console.WriteLine("starting delegate beta:");
beta("Beta");
Console.WriteLine("starting delegate gamma:");
gamma("Gamma");
Console.WriteLine("starting delegate delta:");
delta("Delta"); } }
```

The output of this program will get displayed as:



This sample demonstrates the delegate types. It shows how to map delegates to static and instance methods, and how to combine them (multicast).

Building and Running the Sample Within Visual Studio

To build and run the delegates samples

1. Open the solution (delegates.sln).
2. In Solution Explorer, right-click the Delegates1 project and click Set as StartUp Project.
3. From the Debug menu, click Start Without Debugging.
4. In Solution Explorer, right-click the Delegates2 project and click Set as StartUp Project.
5. From the Debug menu, click Start Without Debugging.
6. Building and Running the Sample from the Command Line

To build and run the delegates samples from the Command Line:

1. Use the cd command to change to the Delegates1 directory.

Type the following:

```
csc book1.cs
book1
```

Use the cd command to change to the Delegates2 directory.

Type the following:

```
csc book2.cs
book2
```

13.14 Delegate Declarations

A delegate-declaration is a type-declaration that declares a new delegate type.

delegate-declaration:

- Attributesopt delegate-modifiersopt delegate result-type
- Identifier (formal-parameter-listopt) ;
- Delegate-modifiers:
- Delegate-modifier
- Delegate-modifiers delegate-modifier delegate-modifier:
- new
- public
- protected
- internal
- private

It is an error for the same modifier to appear multiple times in a delegate declaration.

The new modifier is permitted only on delegates declared within another type. It specifies that the delegate hides an inherited member by the same name. The public, protected, internal, and private modifiers control the accessibility of the delegate type. Depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted.

The formal-parameter-list identifies the signature of the delegate, and the result-type indicates the return type of the delegate. The signature and return type of the delegate must exactly match the signature and return type of any method that the delegate type encapsulates. Delegate types in C# are name equivalent, but they are not structurally equivalent. Two different delegates types that have the same signature and return type are considered different delegate types.

C# offers special syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class

or instance. In particular, it is possible to access members of the System.Delegate type via the usual member access syntax.

13.15 Combinable Delegate Types

Delegate types are classified into two kinds: combinable and non-combinable. A combinable delegate type must satisfy the following conditions:

1. The declared return type of the delegate must be void.
2. None of the parameters of the delegate type can be declared as output parameters.

A run-time exception occurs if an attempt is made to combine two instances of a non-combinable delegate types unless one or the other is null.

13.16 Delegate Instantiation

Although delegates behave much like other classes, C# provides special syntax for instantiating a delegate instance. A delegate-creation-expression is used to create a new instance of a delegate. The newly created delegate instance then refers to either:

1. The static method referenced in the delegate-creation- expression, or the target object (which cannot be null) and instance method referenced in the delegate-creation-expression; or
2. Another delegate.

Once instantiated, delegate instances always refer to the same target object and method.

13.17 Multi-Cast Delegates

Delegates can be combined using the addition operator, and one delegate can be removed from another using the subtraction operator. A delegate instance created by combining two or more (non-null) delegate instances is called a multicast delegate instance. For any delegate instance, the invocation list of the delegate is defined as the ordered list of non-multicast delegates that would be invoked if the delegate instance were invoked. More precisely:

1. For a non-multicast delegate instance, the invocation list consists of the delegate instance itself.
2. For a multi-cast delegate instance that was created by combining two delegates, the invocation list is the formed by concatenating the invocation lists of the two operands of the addition operation that formed the multi-cast delegate.

13.18 Delegate Invocation

C# provides special syntax for invoking a delegate. Invocation of a multi-cast delegate proceeds by the invocation of each of the delegates on the invocation list, in order. Each call is passed the same set of arguments. If the delegate includes reference parameters, each method invocation will occur with a reference to the same variable.Changes to that variable by any method in the invocation list will be 'seen' by any later methods in the invocation list.

If an exception occurs during processing of the invocation of a multicast delegate, and the exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods later in the invocation list are not invoked.

13.20 Interfaces

An interface is a declaration of a collection of class members. They do not contain implementation, instead they indicates that a class that implements the interface must provide those members. Consider this interface:

```
public interface INamed
{
   String Name {get;}
   void PrintName();
}
```

This indicates that a class that implements this interface must have a read-only property called Name and a method called PrintName(). For example:
Listing 13.23

```
public class NamedClass : INamed
{
   protected String name = "NamedClass";
   public String Name
   {
      get{return name;}
   }
   public void PrintName()
   {
      Console.WriteLine("Name is {0}", name);
   }
}
```

The NamedClass class can be used like this:

```
NamedClass c = new NamedClass(); c.PrintName(); INamed ic = c;
ic.PrintName();
```

the output from PrintName() is the same in both the cases. To prevent this, that is, to ensure that the interface members are only accessed through an interface,for which the method name has to be qualified with the interface name.
Note that the interface members are not public, that is, they are not accessible through a NamedClass reference. However, since they are qualified with the interface name, they are accessible through an interface:

```
INamed i = new NamedClass();
i.PrintName(); // compiles OK
```

A class can implement more than one interface, so if you have another class that implements INamed and IViewable the code can just ask for the behavior determined by one of these interfaces, for example INamed:

Listing 13.25

```
public class NVOne : INamed, IViewable {
/* implement INamed and IViewable members*/
} public class NVTwo : INamed, IViewable {
/* implement INamed and IViewable members*/ }
public class A {
   private void f(INamed n)
   {
      n.PrintName();
   }
   public void g()
   {
      NVOne n1 = new NVOne();
      f(n1);
      NVTwo n2 = new NVTwo();
      f(n2);
   }
}
```

Interfaces allow you to select the behavior that you want, in this code the method A.g() creates two objects, one each of two different types. These two objects are passed to the method A.f() that uses just the INamed behavior of each of these objects. Interface programming like this is very powerful; in fact it is the basis of COM. Classes can have a base class besides the indication for implementing interfaces. So, if NamedBase has the code for the INamed interface members:

Listing 13.26

```
public class NamedBase {
   protected String name = "empty";
   public String Name
   {
      get{return name;}
   }
   public void PrintName()
   {
      Console.WriteLine("Name is {0}", name);
   }
}
```

This can be used with classes that implement the INamed interface, which will inherit the NamedBase implementation:

Listing 13.27

```
public class One : NamedBase, INamed
{
   public One()
   {
      name = "One";
```

```
    }
}
public class Two : NamedBase, INamed
{
    public Two()
    {
        name = "Two";
    }
}
```

These classes could be used like this:

```
INamed[] namedObjects = {new One(), new Two()};
foreach (INamed o in namedObjects)
    o.PrintName();
```

This creates an array with instances of the two object types. The foreach construct is used to access all the members in the array, and since the members are INamed, the code can call PrintName() irrespective of the actual type of the object.

13.21 Delegates vs. Interfaces

Delegates and interfaces are similar in that they enable separation of specification and implementation. Multiple independent authors can produce implementations that are compatible with a specific interface. Similarly, a delegate specifies the signature of a method, and authors can write methods that are compatible with the delegate specification.The following points are noteworthy with regard to the topic at hand:

Delegates are useful when:

1. A single method is being called.
2. A class may want to have multiple implementations of the method specification.
3. It is desirable to allow using a static method to implement the specification.
4. An event-like design pattern is desired (for more information, see the Events Tutorial).
5. The caller has no need to know or obtain the object that the method is defined on.
6. The specific implementation is to be handed out to only a few select components. 7. Easy composition is desired.

Interfaces are useful when:

1. The specification defines a set of related methods that will be called. 2. A class typically implements the specification only once. 3. The caller of the interface wants to cast to or from the interface type to obtain other interfaces or classes.

13.22 Name Spaces

In Lesson 4, you saw the 'using System;' directive in the Simple Hello program. This directive allowed you to use members for the System namespace. by the time you complete this lesson you would have understood the 'using' directive and more.

Namespaces are C# program elements designed to help you organize your programs. They also provide assistance in avoiding name clashes between two sets of code. Implementing Namespaces in your own code helps you later when you want to reuse some of your code.

Namespace

C# has a number of readymade methods and classes availabe for use of programmers, which have been encapsulated in one environment i.e. space by name System.The System is a namespace and propritiery of Microsoft. The user can create her/his own namespace and use as independent group. We have been using the statement:

```
System.Console.WriteLine()
```

quite frequently. In this WriteLine() is a function that is a part of class Console. The class Console has a number of other functions available that are related to working on a console like ReadLine etc. The class Console is a part of namespace created by Microsoft and is their proprietary. You can also create a class console exclusively for your application that uses the computer's screen and use in a namespace that is attributed to you. If at a particular place the namespace's name has not been mentioned, the computer presumes it to be System. It is better to write 'using System;' because in that case you will not have to prefix the word System in the above example and can simply write 'Console.WriteLine()'. The following example illustrates this:

Listing 13.28

```
//sharp43.cs
class hema {
   static void Main()
   {
     yyy.alpha();
     alpha();
     hema.alpha();
   }
   public static void alpha()
   {
     System.Console.WriteLine("alpha in hema");
   }
}
namespace Asim_and_Rajan {
     class yyy
     {
       public static void alpha()
       {
         System.Console.WriteLine("alpha");
       }
     }
}
```

The output of this program shall be displayed on console as:

```
C:\>csc c:\sharp\sharp1\sharp43.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp43.cs(6,4): error CS0234: The type or namespace name 'yyy' doe
s not exist in the class or namespace ''
```

Since the class yyy is a part of name space 'Asim and Rajan' but had been called upon in class hema which is by default part of another namespace System, although this name is not written.

Listing 13.29

```
//sharp44.cs
class hema {
   static void Main()
   {
     vkjain.yyy.alpha();
     alpha();
     hema.alpha();
   }
   public static void alpha()
   {
     System.Console.WriteLine("alpha in hema");
   }
}
namespace vkjain {
   class yyy
   {
     public static void alpha()
     {
       System.Console.WriteLine("alpha");
     }
   }
}
```

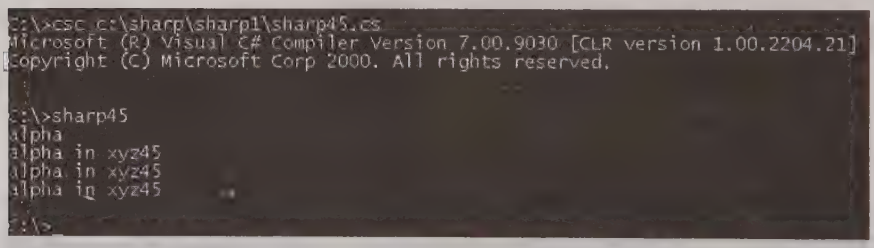
The output of this program shall be displayed on console as:

```
C:\>csc c:\sharp\sharp1\sharp44.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.


C:\>sharp44
alpha
alpha in xyz44
alpha in xyz44

C:\>
```

Sice we have changed the function call from yyy.alpha(); to vkjain.yyy.alpha();the program had been properly compiled.

Listing 13.30

```
//sharp45.cs
namespace chatterjee {
  class hema
  {
   static void Main()
   {
     schatterjee.yyy.alpha();
     alpha();
     hema.alpha();
     jain.hema.alpha();
   }
   public static void alpha()
   {
     System.Console.WriteLine("alpha in hema");
   }
} }
namespace schatterjee {
   class yyy
   {
     public static void alpha()
     {
       System.Console.WriteLine("alpha");
     }
   }
}
```



The statement schatterjee.yyy.alpha(); means calling function yyy.alpha() of namespace schatterjee , and simply alpha(); means calling function alpha() from namespace chatterjee. Similarly the function call hema.alpha(); means function alpha of class hema and namespace chatterjee (although not specified).The statement chatterjee.hema.alpha(); has also the same meaning.

Listing 13.31

```
//sharp46.cs
class hema
```

```
    {
    static void Main()
    {
      Console.WriteLine("alpha in hema");
    }
}
```

```
C:\>csc C:\sharp\sharp1\sharp46.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

sharp\sharp1\sharp46.cs(6,4): error CS0234: The type or namespace name 'Console'
 does not exist in the class or namespace '

C:\>
```

In this short example, we have used the word Console but haven't specified the namespace System of which Console is a part, and that caused the error message. Therefore, add the command line using System; as the first line as has been done in example below:

Listing 13.32

```
//sharp47.cs
using System;
class hema {
   static void Main()
{
   Console.WriteLine("alpha in hema");
} }
```

The output of this program shall be displayed on console as:

```
C:\>sharp47
alpha in hema
```

Listing 13.33

```
//sharp48.cs
using System;
class xyz48 {
   static void Main()
   {
     yyy.alpha();
   }
}
   namespace vkjain
   {
     class yyy
     {
       public static void alpha()
```

```
        {
          Console.WriteLine("alpha");
        }
      }
    }
```

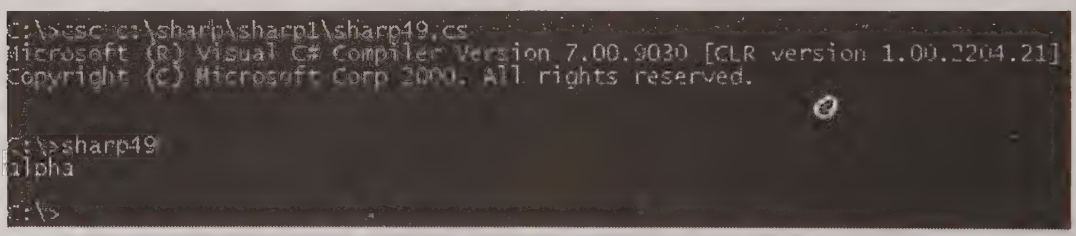
The output of this program shall be displayed on console as:

In this example, the function alpha() has been called as yyy.alpha() but the function alpha() exists in namespace vkjain. Since the name of class has not been added in this function call i.e. yyy.alpha();. the compiler has taken yyy to be namespace.

Listing 13.34

```
//sharp49.cs
using System; using vkjain; class xyz49 {
   static void Main()
{
   yyy.alpha();
} }
   namespace vkjain
{
   class yyy
{
   public static void alpha()
{
  Console.WriteLine("alpha");
} } }
```

The output of this program shall be displayed on console as:



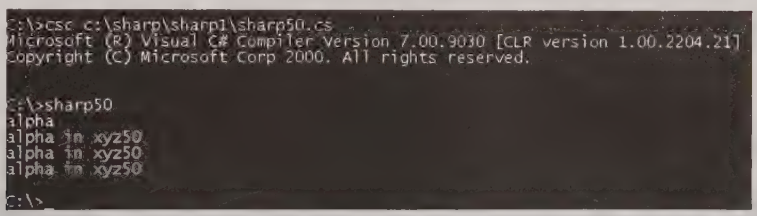
In this case we have added the statement for using namespace ++vkjain. Due to the function call yyy.alpha() compiler first tries to search the namespace yyy which is not there. Then it resorts to using vkjain namespace, in which it finds alpha() function in class yyy.

Listing 13.35

```
Editors //sharp50.cs
using System;
using jain;
using vkjain;
namespace jain {
   class xyz50
   {
     static void Main()
     {
```

```
        yyy.alpha();
        alpha();
        xyz50.alpha();
        xyz50.alpha();
      }
      public static void alpha()
      {
        System.Console.WriteLine("alpha in xyz50");
      }
    }
}
namespace vkjain {
    class yyy
    {
      public static void alpha()
      {
        System.Console.WriteLine("alpha");
      }
    }
}
```

The output of this program shall be displayed on console as:

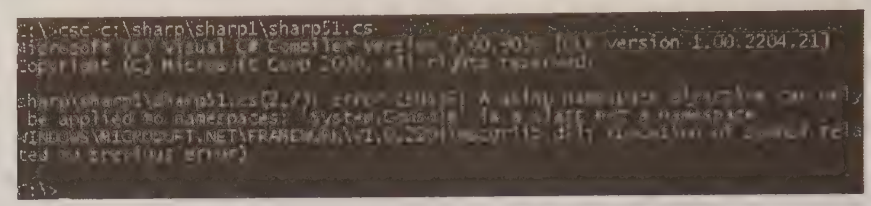


The above program has a number of using statements. The compiler tries each one of them to perform a particular task and issues an error message if is unable to find the required method (function) in any one of them.

Listing 13.36

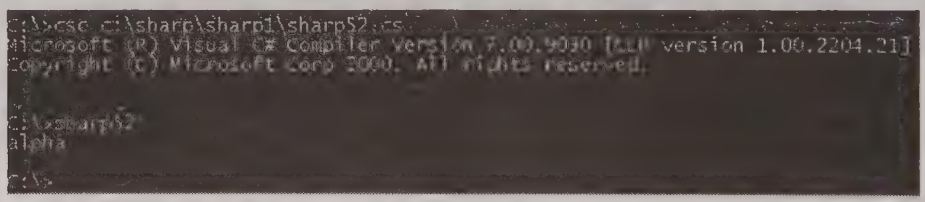
```
//sharp51.cs
using System.Console; class xyz51 {
    static void Main()
    {
      WriteLine("alpha in xyz51");
    }
}
```

The output of this program shall be displayed on console as:



```
//sharp52.cs
class  xyz52
  {
   static void Main()
   {
     jain.vkjain.yyy.alpha();
   }
 }
namespace jain {
    namespace vkjain
    {
      class yyy
       {
         public static void alpha()
        {
            System.Console.WriteLine("alpha");
        }
      }
    }
}
```

The output of this program shall be displayed on console as:



In this example,the hierarchical concept of using namespaces has been employed. We have specified namespace vkjain inside namespace jain and called a function abc() from main function, which exists in class yyy of ultimate namespace vkjain. This is hierarchically specified as:

```
jain.vkjain.yyy.alpha();
```

```
Listing 13.37
// Namespace Declaration
// name1.cs
using System;
// About  Namespace
namespace csharp {
    // starting the  class
    class Namespace1 {
```

```
        // Main begins program execution.
        public static void Main() {
            // Write to console
            Console.WriteLine("New C#  Namespace.");
        }
    }
}
```

The output of this program will be as shown below:

```
C:\>name1
New C#  Namespace.
```

We declare the new namespace by putting the word 'namespace' in front of 'csharp'. Curly braces surround the members inside the 'csharp' namespace.

Listing 13.38

```
// Namespace Declaration
using System;
// The C# 1 Namespace
namespace csharp {
    namespace lesson {
        // Program start class
        class Namespace2 {
            // Main begins program execution.
            public static void Main() {
                // Write to console
                Console.WriteLine("New C#   Namespace.");
            }
        }
    }
}
```

The output of this program shall be :

```
C:\>name2
New C#   Namespace.
```

Namespaces allow you to create a system. You put the more general names at the top of the hierarchy and the more specific down through. This hierarchical system can be represented by nested namespaces by placing code in different sub-namespaces, codes can be organized.

Listing 13.4: Calling Namespace Members

```
// name4.cs
using System;
namespace csharp {
    // nest in  namespace
    namespace lesson {
        class myExample1 {
            public static void myPrint1() {
       Console.WriteLine(
" Example 1 of calling another namespace member.");
            }
```

```
        }
    }
    // Program start class
    class NamespaceCalling {
        // Main begins program execution.
        public static void Main() {
            // Write to console
            lesson.myExample1.myPrint1();
            csharp.lesson.myExample2.myPrint2();
        }
    }
}
// same namespace as nested namespace above
namespace csharp.lesson {
    class myExample2 {
        public static void myPrint2() {
  Console.WriteLine("Example 2 of calling another namespace member.");
        }
    }
}
```

The output of this program shall be:

```
C:\>name4
 Example 1 of calling another namespace member.
Example  2 of calling another namespace member.
```

Listing 13.4 provides an example of calling namespace members with fully qualified names. A fully qualified name contains every language element from the namespace name down to the method call. At the top of the listing there is a nested namespace 'lesson' within the 'csharp' namespace with class 'myExample1' and method 'myPrint1'. Main() calls this method with the fully qualified name of 'lesson.myExample1.myPrint()'. Since Main() and the lesson namespace are located in the same namespace, using "csharp" in the fully qualified name is unnecessary.

At the bottom of the Listing 13.4 is an addition to the 'csharp.lesson' namespace. The classes 'myExample1' and 'myExample2' both belong to the same namespace. Additionally, they could be written in separate files and still they can belong to the same namespace. In Main(), the 'myPrint2' method is called with the fully qualified name 'csharp.lesson.myExample2.myPrint2()'. It must use 'csharp' in the fully qualified name, because 'myExample2' is declared outside the bounding braces.

Note that we have used different names for the two classes 'myExample1' and 'myExample2'. This was necessary because every namespace member of the same type must have a unique name. Since both are in the same namespace we need to eliminate any ambiguity about which class to use. The methods 'myPrint1' and 'myPrint2' having different names makes the lesson a little easier to follow. They could have had the same name with no effect, because their classes are different, thus avoiding any ambiguity.

To call methods without typing their fully qualified name, one may implement the 'using' directive. In some of our Listings, we see two 'using' directives. The first, 'using System', is the same 'using' directive you have seen in every program in this lesson. It allows you to type the method names of members of the 'System' namespace without typing the word 'System' every time. In myPrint(), 'Console' is a class member of the 'System' namespace with the method 'WriteLine'. It's fully qualified name is 'System.Console.WriteLine(...)'.

Similarly, the using directive 'using csharp.lesson' allows us to implement members of the 'csharp.lesson' namespace without typing the fully qualified name. This is why we can type 'myExample.myPrint()'. Without the 'using' directive, we would have to type 'csharp.lesson.myExample.myPrint()' every time we wanted to implement that method.

Sometimes you may encounter a long namespace and may wish to shorten it and still avoid name clashes with similarly named methods. Alies can be created with the alias directive using csTut = csharp.lesson.myExample. Now the expression 'csTut' can be used anywhere in place of 'csharp.lesson.myExample'. We use it in Main().

Also in Main() is a call to the 'AliasDirective' classes' 'myPrint' method. This is the same name as the 'myExample' classes' 'myPrint' method.Both of these methods can be called in the same method call is because the 'myExample' classes' 'myPrint' method is qualified with the 'csTut' alias. So the compiler knows exactly which method is to be executed. Had we, by mistake omitted 'csTut' from the method call, the compiler would have set up the 'AliasDirective' classes' 'myPrint' method to run twice.

What would have happened if we had not created the alias directive and instead added a 'using csharp.lesson.myExample' in it's place and then called myPrint(). The compiler would have generated an error because it cannot figure out whether to call 'csharp.lesson.myExample.myPrint()' or 'AliasDirective.myPrint()'. This illustrates the merit of using namespaces. It will help avoid conflicts in your code.

So far, all we have shown in our namespaces are classes. However, namespaces can hold other types such as :

- Classes
- Structures
- Interfaces
- Enumerations
- Delegates

13.23 Types of Namespaces

C# programs are organized using namespaces. Namespaces are used both as an 'internal' organization system for a program, and as an 'external' organization system-a way of presenting program elements that are exposed to other programs.

Using directives are provided to facilitate the use of namespaces.

13.23.1 Compilation Units

A compilation-unit defines the overall structure of a source file. A compilation unit consists of zero or more using- directives followed by zero or more namespace-member- declarations.

compilation-unit:
using-directivesopt attributesopt namespace-member-declarationsopt

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The using-directives of a compilation unit affect the attributes and namespace-member-declarations of that compilation unit, but have no effect on other compilation units.

The attributes of a compilation unit permit the specification of attributes for the target assembly and module. Assemblies and modules act as physical containers for types. An assembly may consist of several physically separate modules.

The namespace-member-declarations of each compilation unit of a program contribute members to a single declaration space called the global namespace. For example:

```
File A.cs: class A {} File B.cs: class B {}
```

The two compilation units contribute to the single global namespace. In this case they declare two classes with the fully qualified names A and B. Because the two compilation units contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

13.24 Namespace Declarations

A namespace-declaration consists of the keyword namespace, followed by a namespace name and body, optionally followed by a semicolon.

Listing 13.43

```
namespace-declaration:
namespace   qualified-identifier   namespace-body   ;opt
qualified-identifier:
identifier qualified-identifier       identifier
namespace-body:
{   using-directivesopt   namespace-member-declarationsopt   }
```

A namespace-declaration may occur as a top-level declaration in a compilation-unit or as a member declaration within another namespace-declaration. When a namespace-declaration occurs as a top-level declaration in a compilation-unit, the namespace becomes a member of the global namespace. When a namespace- declaration occurs within another namespace-declaration, the inner namespace becomes a member of the outer namespace. In either case, the name of a namespace must be unique within the containing namespace.

Namespaces are implicitly public and the declaration of a namespace cannot include access modifiers. Within a namespace-body, the optional using-directives import the names of other namespaces and types, allowing them to be referenced directly instead of through qualified names. The optional namespace-member-declarations contribute members to the declaration space of the namespace. Note that all using-directives must appear before any member declarations.

The qualified-identifier of a namespace-declaration may be a single identifier or a sequence of identifiers separated by "." tokens. The latter form permits a program to define a nested namespace without lexically nesting several namespace declarations. For example, consider the two namespace declarations that follow:

Listing 13.44

```
namespace N1.N2 {
     class A {}
     class B {}
} is semantically equivalent to namespace N1 {
     namespace N2
     {
          class A {}
          class B {}
     }
}
```

Namespaces are open-ended, and two namespace declarations with the same fully qualified name contribute to the same declaration space . In the example

```
namespace N1.N2 {
     class A {}
} namespace N1.N2 {
     class B {}
}
```

These two namespace declarations contribute to the same declaration space, in this case declaring two classes with the fully qualified names N1.N2.A and N1.N2.B. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

13.25 Using Directives

Using directives facilitate the use of namespaces and types defined in other namespaces. Using directives impact the name resolution process of namespace-or-type-names and simple-names, but unlike declarations, using directives do not contribute new members to the underlying declaration spaces of the compilation units or to the namespaces within which they are used.

using-directives:using-directive using-directives using-directive using-directive: using-alias-directive using-namespace-directive

A using-alias-directive introduces an alias for a namespace or type.

A using-namespace-directive imports the type members of a namespace. The scope of a using-directive extends over the namespace-member- declarations of its immediately containing compilation unit or namespace body. The scope of a using-directive specifically does not include its peer using-directives. Thus, peer using- directives do not affect each other, and the order in which they are written is insignificant.

13.26 Using Namespace Directives

A using-namespace-directive imports the types contained in a namespace into the immediately enclosing compilation unit or namespace body, enabling the identifier of each type to be used without qualification.

```
using-namespace-directive: using   namespace-name   ;
```

Within member declarations in compilation unit or namespace body that contains a using-namespace-directive, the types contained in the given namespace can be referenced directly. For example:

Listing 13.45

```
namespace N1.N2 {
     class A {}
} namespace N3 {
     using N1.N2;
     class B: A {}
}
```

Here, within member declarations in the N3 namespace, the type members of N1.N2 are directly available, and thus class N3.B derives from class N1.N2.A.

A using-namespace-directive imports the types contained in the given namespace, but specifically does not import nested namespaces. In the example

Listing 13.46

```
namespace N1.N2 {
     class A {}
} namespace N3 {
     using N1;
     class B: N2.A {}          // Error, N2 unknown
}
```

the using-namespace-directive imports the types contained in N1, but not the namespaces nested in N1. Thus, the reference to N2.A in the declaration of B is in error because no members named N2 are in scope.

13.27 Namespaces and Assemblies

Namespaces provide a logical organizational system. Namespaces as mentioned earlier, are used both as an 'internal' organization system for a program, and as an 'external' organization system. Assemblies are used for physical packaging and deployment. An assembly acts as a container for types. An assembly may contain types, the executable code used to implement these types, and references to other assemblies.

There are two main kinds of assemblies: applications and libraries. Applications have a main entry point and usually have a file extension of .exe; libraries do not have a main entry point, and usually have a file extension of .dll. To demonstrate the use of namespaces and assemblies, this section revisits the 'hello, dreamtech' program presented earlier, and splits it into two pieces: a library that provides messages and a console application that displays them.

The library will contain a single class named HelloMessage. The example

Listing 13.47

```
// HelloLibrary.cs namespace Microsoft.CSharp.Introduction {
    public class HelloMessage
    {
        public string Message {
            get {
                return "hello, dreamtech";
            }
        }
    }
}
```

shows the HelloMessage class in a namespace named Microsoft.CSharp.Introduction. The HelloMessage class provides a read-only property named Message. Namespaces can nest, and the declaration

namespace Microsoft.CSharp.Introduction {_} is shorthand for several levels of namespace nesting: namespace Microsoft {

Listing 13.48

```
    namespace CSharp
    {
        namespace Introduction
        {_}
    }
}
```

The next step in the componentization of 'hello, dreamtech' is to write a console application that uses the HelloMessage class. The fully qualified name for the class- Microsoft. CSharp.Introduction.HelloMessage-could be used, but this name is quite long and unwieldy. An easier way is to use a using namespace directive, which makes it possible to use all of the types in a namespace without qualification. The example

Listing 13.49

```
// HelloApp.cs
 using Microsoft.CSharp.Introduction;
 class HelloApp {
    static void Main() {
        HelloMessage m = new HelloMessage();
        System.Console.WriteLine(m.Message);
    }
}
```

shows a using namespace directive that refers to the Microsoft.CSharp.Introduction namespace.The occurrences of HelloMessage are shorthand for Microsoft.CSharp. Introduction. HelloMessage.

C# also enables the definition and use of aliases. A using alias directive defines an alias for a type. Such aliases can be useful in situation in which name collisions occur between two libraries, or when a small number of types from a much larger namespace are being used.

The example using MessageSource = Microsoft.CSharp.Introduction.HelloMessage; shows a using alias directive that defines MessageSource as an alias for the HelloMessage class.

The code we have written can be compiled into a library containing the class HelloMessage and an application containing the class HelloApp. The details of this compilation step might differ based on the compiler or tool being used. Using the command-line compiler provided in Visual Studio 7.0, the correct invocations are

csc /target:library HelloLibrary.cs which produces the class library HelloLibrary.dll and csc /reference:HelloLibrary.dll HelloApp.cs which produces the application HelloApp.exe.

13.28 Versioning

Versioning is the process of evolving a component over time in a compatible manner. A new version of a component is source compatible with a previous version if the code that depends on the previous version can, when recompiled, work with the new version. In contrast, a new version of a component is binary compatible if a program that depended on the old version can, without recompilation, works with the new version.

Most languages do not support binary compatibility at all, and many do little to facilitate source compatibility. In fact, some languages contain flaws that make it impossible, in general, to evolve a class over time without breaking at least some client code.

As an example, consider the situation of a base class author who ships a class named Base. In the first version, Base contains no F method. A component named Derived derives from Base, and introduces an F. This Derived class, along with the class Base that it depends on, is released to customers, who deploy to numerous clients and servers.

Listing 13.50

```
// Author A namespace A {
     public class Base  // version 1
     {
     }
} // Author B namespace B {
     class Derived: A.Base
     {
          public virtual void F() {
               System.Console.WriteLine("Derived.F");
          }
     }
} So far, so good. But now the versioning trouble begins. The author of
Base produces a new version, and adds its own F method.
// Author A namespace A {
     public class Base        // version 2
     {
```

```
            public virtual void F() {
// added in version 2
                System.Console.WriteLine("Base.F");
            }
        }
}
```

This new version of Base should be both source and binary compatible with the initial version. (If it weren't possible to simply add a method then a base class could never evolve.) Unfortunately, the new F in Base makes the meaning of Derived's F unclear. Did Derived mean to override Base's F? This seems unlikely, since when Derived was compiled, Base did not even have an F. Further, if Derived's F does override Base's F, then it must adhere to the contract specified by Base-a contract that was unspecified when Derived was written? In some cases, this is impossible. For example, the contract of Base's F might require that overrides of it always call the base. Derived's F could not possibly adhere to such a contract.

C# addresses this versioning problem by requiring developers to clearly state their intent. In the original code example, the code was clear, since Base did not even have an F. Clearly, Derived's F is intended as a new method rather than an override of a base method, since no base method named F exists.

Listing 13.51

```
// Author A
namespace A {
    public class Base
    {
    }
} // Author B
namespace B {
    class Derived: A.Base
    {
        public virtual void F() {
            System.Console.WriteLine("Derived.F");
        }
    }
```

If Base adds an F and ships a new version, then the intent of a binary version of Derived is still clear-Derived's F is semantically unrelated, and should not be treated as an override. However, when Derived is recompiled, the meaning is unclear-the author of Derived may intend its F to override Base's F, or to hide it. Since the intent is unclear, the compiler produces a warning, and by default makes Derived's F hide Base's F. This course of action duplicates the semantics for the case in which Derived is not recompiled. The warning that is generated alerts Derived's author to the presence of the F method in Base. If Derived's F is semantically unrelated to Base's F, then Derived's author can express this intent-and, in effect, turn off the warning-by using the new keyword in the declaration of F.

Listing 13.52

```
// Author A
namespace A {
     public class Base              // version 2
     {
          public virtual void F() { // added in version 2
               System.Console.WriteLine("Base.F");
          }
     }
} // Author B
namespace B {
     class Derived: A.Base     // version 2a: new
     {
          new public virtual void F() {
               System.Console.WriteLine("Derived.F");
          }
     }
}
```

On the other hand, Derived's author might investigate further, and decide that Derived's F should override Base's F. This intent can be specified by using the override keyword, as shown below.

Listing 13.53

```
 // Author A
namespace A {
     public class Base                         // version 2
     {
          public virtual void F() { // added in version 2
               System.Console.WriteLine("Base.F");
          }
     }
} // Author B
namespace B {
     class Derived: A.Base     // version 2b: override
     {
          public override void F() {
               base.F();
               System.Console.WriteLine("Derived.F");
          }
     }
}
```

The author of Derived has another option, that is, to change the name of F, thus completely avoiding the name collision. Though this change would break source and binary compatibility for Derived, the importance of this compatibility varies depending on the scenario. If Derived is not exposed to other programs, then changing the name of F is a good idea, as it would improve the readability of the program-there would no longer be any confusion about the meaning of F.

13.29 Uses of Versioning

This section demonstrates versioning in C# through the use of the override and new keywords. Versioning maintains compatibility between base and derived classes as they evolve.

The C# language is designed such that versioning between base and derived classes in different libraries can evolve and maintain backwards compatibility. This means, for example, that the introduction of a new member in a base class with the same name as a member in a derived class is not an error. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that simply hides a similarly named inherited method.

In C#, methods are by default not virtual. To make a method virtual, the virtual modifier has to be used in the method declaration of the base class. The derived class can then override the base virtual method by using the override keyword or hide the virtual method in the base class by using the new keyword. If neither the override keyword nor the new keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class. The following example shows these concepts in action.

Listing 13.54

Example 000: // Versioning\versioning.cs

```
001: public class MyBase
002: {
003:     public virtual string Meth1()
004:     {
005:         return "MyBase-Meth1";
006:     }
007:     public virtual string Meth2() 008:     {
009:         return "MyBase-Meth2";
010:     } 011:     public virtual string Meth3() 012:     {
013:         return "MyBase-Meth3";
014:     }
015: }
016:
017: class MyDerived : MyBase
018: {
019:     public override string Meth1()
020:     {
021:         return "MyDerived-Meth1";
022:     }
023:     public new string Meth2()
024:     {
025:         return "MyDerived-Meth2";
026:     }
027:     public string Meth3() // Issues a warning to alert the
programmer
028:                           // that the method hides the inherited
029:                           // member MyBase.Meth3().
030:     {
031:         return "MyDerived-Meth3";
```

```
032:     }
033:
034:     public static void Main()
035:     {
036:         MyDerived mD = new MyDerived();
037:         MyBase mB = (MyBase) mD;
038:
039:         System.Console.WriteLine(mB.Meth1());
040:         System.Console.WriteLine(mB.Meth2());
041:         System.Console.WriteLine(mB.Meth3());
042:     }
043: }
```

The output of this program shall get displayed as:

```
C:\>csc \xml.cs /doc:xml.xml
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>xml
```

Explanation

Line 019 overrides the virtual method Meth1 using the override keyword. Line 023 explicitly hides the virtual method Meth3 using the new keyword. Line 027 hides the virtual method Meth3 because a keyword is not specified. A warning is issued to alert the programmer that a new or an override keyword should be specified. Hiding a base class member from a derived class isn't an error in C#. This feature enables you to make changes in the base class without breaking other libraries that inherit this base class.

For example, at some point you could have the following classes:

Listing 13.55

```
class Base {} class Derived: Base {
    public void F() {}
}

At some later point, the base class could evolve to add a void method
F() as follows:

class Base {
    public void F() {}
} class Derived: Base {
    public void F() {}
}
```

13.30 A Component in C#

Here is how a simple string component appears in C#:

Listing 13.56: Component in C# (CompCS.cs)

```
using System;
namespace CompCS {
   public class StringComponent {
      private string[] StringsSet;
      public StringComponent() {
         StringsSet = new string[] {
            "C# String 0",
            "C# String 1",
            "C# String 2",
            "C# String 3"
         };
      }
      public string GetString(int index) {
         if ((index < 0) || (index >=
            StringsSet.Length)) {
            throw new IndexOutOfRangeException();
         }
         return StringsSet[index];
      }
      public int Count {
         get { return StringsSet.Length; }
      }
   }
}
```

As mentioned above, we use the namespace statement to create a new namespace to encapsulate the classes we will be creating:

```
namespace CompCS {_}
```

This namespace may be nested and may be split between multiple files: A single source code file may also contain multiple non- nested namespaces. A containing namespace is required, since all C# code must be contained in a class.

```
public class StringComponent {_}
```

This statement means that instances of StringComponent will now be created by the runtime and managed in the Garbage Collected heap. The class constructor - which executes each time a new instance of the class is created - has the same name as the class and doesn't have a return type.

```
public StringComponent() {_}
```

Since C# uses only managed types, we don't have to do anything special - as we had to do with Managed Extensions - when declaring variables.

Here's the GetString method, which takes an integer and returns a string:

```
public string GetString(int index) {

   return StringsSet[index];
}
```

Note the throw statement in the GetString method:throws new IndexOutOfRangeException(); This statement creates - and throws - a runtime-based exception handling object of type IndexOutOfRangeException.
Another way of returning strings would be to use an indexer instead of the GetString method:

Listing 13.57

```
public string this[int index] {
   if ((index < 0) || (index >= Count)) {
      throw new IndexOutOfRangeException();
   }
   return StringsSet[index];
}
```

An indexer would allow the client to use a much more natural syntax of:
StringSetVar[index] VB, however, doesn't support indexers so, in the interest of keeping the component code the space for each of the three languages, we have chosen to use a distinct GetString method.

Finally, we create the read-only property Length:

```
public int Count {
   get { return StringsSet.Length; }
}
```

The compile process for a C# component is only a little more complicated than for a stand-alone program:
csc /out:..\Bin\CompCS.dll /target:library CompCS.cs As with the C++ component, we use the /out switch to put the compiled component in the "..\Bin" relative subdirectory for convenience. Likewise, we need the /target:library switch to actually create a DLL rather than an executable with a .dll file extension.

Indexers

To include access to your class, just like an array the indexer feature of C# may be used.

Basically, the syntax looks like this:

```
attributes modifiers declarator {declarations}
```

A sample implementation could be
Listing 13.58

```
public string this [int nIndex] {
  get {......}
  set {......}
}
```

This indexer returns or sets a string at a given index. It has no attributes,but uses the public modifier. The declarator part consists of type string and this is to denote the class's indexer. The implementation rules for get and set are the same as for properties.(You can drop either one.) There is one diffrence,though: You are almost free in defining the parameter list in the square brackets .The restrictions are that you must specify at least one parameter, and ref and out modifiers are not allowed.

The This keyword warrants an explanation.Indexers do not have user-defined names, and this denotes the indexer on the default interface.If your class implements multiple interfaces, you can add more indexers denoted with InterfaceName.this.

To demonstrate the use of an indexer ,we willI create a small class that is capable of resolving a hostname to an IP address-or, as is the case for www.microsoft.com,resolving to a list of IP addresses. This list is accessible via an indexer , and you can take a look at the implementation in Listing 13.8.

Listing 13.59

```
//input6.cs

using System;
using System.Net;
class ResolveDNS {
       IPAddress[ ]  m_arrIPs;
       public void Resolve(string strHost)
       {
            IPHostEntry  iphe = DNS.GetHostByName(strHost);
            m_arrIPs = iphe.AddressList;
       }
       public IPAddress this [int nIndex]
       {
            get
            {
                 return  m_arrIPs[nIndex];
            }
       }
       public int Count
       {
             get  { return m_arrIPs.Length; }
       }
}
class DNSResolverApp {
        public static void Main()
        {
             ResolveDNS myDNSResolver = new ResolveDNS();
             myDNSResolver.Resolve("www.microsoft.com");
              int nCount = myDNSResolver.Count;
       Console.WriteLine("Found {0} IP's for hostname", nCount);
              for (int i=0; i < nCount; i++)
                    Console.WriteLine(myDNSResolver[i]);
         }
}
```

To resolve the hostname, we use the DNS class that is part of the System.Net namescape. However, because this namescape is not contained in the core library, we had to reference the library in our compiler statement:

```
csc /r:System.Net.dll /out:resolver.exe dnsresolve.cs
```

The resolver code is straightforward. In the Resolve method,the code calls the static GetHostByName method of the DNS class, which returns an IPHostEntryobject.This object, in turn, contains the array we are looking for-the AddressList array.Before exiting the Resolve method , we store a copy the AddressList array(objects of type IPAddress are stored inside it) locally in the object's instance member m_arrIPs.

With the array now populated, the application code can enumerate the IPaddresses in lines 37 and 38 by using the indexer implemented in the class ResolveDNS. (There is more information about for statements in Chapter6,"Control Statements.")Because there is no way to modify the IP addresses, only get is implemented for the indexer.For simplicity's sake, let us leave out- of-bounds checking to the array.

The output of this program shall be:

Listing 13.60

```
input9.cs using System;
internal class Triangle {
       protected int m_a, m_b, m_c;
       public Triangle(int a, int b, int c)
       {
            m_a = a;
            m_b = b;
            m_c = c;
       }

       public virtual double Area()
       {
            // Heronian formula
            double s = (m_a + m_b + m_c) / 2.0;
            double dArea = Math.Sqrt(s*(s-m_a)*(s-m_b)*(s-m_c));
            return dArea;
       }
}
internal class Prism:Triangle {
     private int m_h;
     public Prism(int a, int b, int c, int h):base(a,b,c)
     {
        m_h = h;
     }
     public override double Area ()
     {
         double dArea = base.Area() * 2.0;
         dArea += m_a*m_h + m_b*m_h + m_c*m_h;
         return dArea;
     }
}
class PrismApp {
```

```
        public static void Main()
        {
             Prism prism = new Prism(2,5,6,1);
             Console.WriteLine(prism.Area());
         }
}
```

The output of this program shall be:

```
C:\>input9
22.3674969975976
```

Both the Triangle and the Prism class are now marked as internal. This means that they are accessible only in the current NGWS component. Please remember that the term NGWS component refers to packaging, and not to the component that you may be used to from COM+. The Triangle class has three protected members, which are initialized in the constuctor and used in the Area calculation method. Because these members are protected, Ican access them in the derived class Prism to perform a different Area calculation there.Prism itself adds an additional member m_h, which is private-not even a derived class could access it.

Listing 13.61: input10.cs using System;

```
class NestedIfApp {
      public static int Main(string[ ] args)
      {
      if (args.Length !=1)
      {
          Console.WriteLine("Usage: one argument");
          return 1;  // error level
       }
       char chLetter = args [0] [0];
       if (chLetter >= 'A' )
          if (chLetter <= 'Z' )
          {
               Console.WriteLine("{0} is uppercase",chLetter);
               return 0;
          }
        chLetter = Char.FromString(args[0]);
        if (chLetter >= 'a' && chLetter <= 'z')
           Console.WriteLine("{0} is lowercase",chLetter);
        if (Char.IsDigit((chLetter = args[0] [0])))
           Console.WriteLine("{0} is a digit",chLetter);
        return 0;
      }
```

The output of this program shall be:

```
C:\>input10
Usage: one argument
```

Listing 13.62

```
input11.cs using System;
class FallThrough {
        public static void Main(string[ ]  args)
        {
            if (args.Length != 1) return;
            int nMonth = Int32.Parse( args[0] );
            if (nMonth < 1 \\ nMonth >12) return;
            int nDays = 0;

            switch (nMonth)
            {
                case 2: nDays = 28; break;
                case 4:
                case 6:
                case 9:
                case 11: nDays = 30; break;
                default: nDays = 31;
             }
             Console.WriteLine("{0} days in this month",nDays);
        }
}
```

The output of this program shall be:

Listing 13.63

```
input12.cs using System; using System.IO;
class WhileDemoApp {
     public static void Main()
     {
       StreamReader sr = File.OpenText ("whilesample.cs");
                string strLine = null;
                while (null != (strLine = sr.ReadLine()))
                {
                       Console.WriteLine(strLine);
                }
                sr.Close();
           }
}
```

The output of this program shall be:

Listing 13.64

```
input13.cs using System;
class jumpTest {
       public static void Main()
       {
            try
            {
                 Console.WriteLine("try here");
                 goto_ _leave ;
            }
            finally
            {
                 Console.WriteLine("finally here");
            }

             _ _leave;
             Console.WriteLine("_ _leave here");
       }
}
```

The output of this program will be:

Listing 13.65

```
input14.cs using System;
class CatchIT {
       public static void Main()
       {
            try
            {
                 int  nTheZero = 0;
                 int  nResult = 10 / nTheZero;
            }
            catch(DivideByZeroException  divEx)
            {
                 Console.WriteLine("divide by zero occured!");
            }
            catch (Exception Ex)
            {
                  Console.WriteLine("some other exception");
            }
            finally
            {
            }
        }
}
```

The output of this program will be:

```
C:\>csc \input14.cs
Microsoft (R) Visual C# Compiler version 7.00.9030 [CLR version 1.00.22
Copyright (C) Microsoft Corp 2000. All rights reserved.

INPUT14.CS(12,13): warning CS0168: The variable 'divEx' is declared but
ed
INPUT14.CS(16,13): warning CS0168: The variable 'Ex' is declared but ne
```

The twist here is that it contains multiple catch state-ments.The first one catches the more likely DivideByZeroException exception, whereas the second catch statement deals with all remaining exceptions by catching the general exception.

you must always catch specialized exceptions first,followed by more general exceptions.

13.1 Preprocessor Directives

If you have done c++ or c programming you may be already knowing the meaning of preprocessor directives.

The preprocessor provides several unique functions that are unavailable to other high level languages but rather are normally associated with assembly languages usage. The preprocessor operates much like a text editor under the control of preprocessor statements of directives. Processor directives are coded into the source program before complilation much like "normal" C statements are. Preprocessor directives also differ from normal statements in form. They do not follow the syntax rules for statements. Instead they observe a smaller set of syntax rules, one of which is that all preprocessor statements must begin with the pound character, # in column one. This character is immediately followed by one of the preprocessor command words, which are not reserved words. Directives do not rely on the semicolon for statement termination. It is essential that the reader keep in mind that these directives are not language statements in the normal sense of the word. There can be no function calling, no variable manipulation, and only the # if directive can make use of expression evaluation. Preprocessor statements may only direct that certain sections of code or strings appear in the extended C source preprocessor compiler phase. The directives can be divided into three categories: file inclusion, macro substitution, and compiler control.

13.2 Preprocessor Definition

The preprocessor program is the first program to run in the series of programs necessary to compile a C program. It reads the C source code looking for preprocessor directive. These directives are marked by a pound sign (#) in the first character position of the source code line. These directives control shows how the preprocessor modifies the source code before it goes to the compiler. One common use of preprocessor directives is to specify which computer and operating system to use to run the program. Other directives in the C source code can then include or delete other statements so that the resulting statements match the target computer and operating system. The same source code then can run on many computers and operating systems. Only the short directive file included in the C source code needs to be different.

Listing 13.66

```
//pre1.cs
#define vkjain
class xyz5 {
    public static void Main()
    {

    }
}
```

In this particular example define is one of the preprocessor directive. this creates a token (identifier or variable) vkjain and it has no other value but that of true.

Listing 13.67

```
//pre2.cs
#define vkjain
class xyz5 {
    public static void Main()
{
   #define vkjain
} }
```

The output of this program will be displayed as below:

```
C:\vcsc \vkj\pre2.cs
Microsoft (R) Visual C# Compiler version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.

vkj\pre2.cs(7,4): error CS1032: Cannot define/undefine preprocessor symbols afte
r first token in file
```

The error message is due to the fact that we have attempted not to put the preprocessor directive as the first line of the code. As a rule, the preprocessor has to at the top of the program before any code is written.

Listing 13.68

```
//sharp5a.cs
#define vkjain
class xyz5 {
    public static void Main()
{
    #if vkjain
     System.Console.WriteLine("vk jain");

     #endif
} }
```

The output of this program will be displayed as shown :

vkjain

One thing is noteworthy that we can always modify the if statement by putting a # before it to work exactly as if statement. The value of variable created by the preprocessor can be checked by #if statement.

The #define preprocessor vkjain creates a variable. When the program goes to #if vkjain. If the value of if statement code is true the writeline will print the message.

Listing 13.69

```
//sharp5.cs
class xyz5 {
   public static void Main()
{
   #if vkjain
   System.Console.WriteLine("vk jain");
   #else
   System.Console.WriteLine("hema");
   #endif
} }
```

The output of this program will be displayed as shown :

hema

In this program we have not included the preprocessor directive so that no varaible is created and value of if statement is thus false and program prints the message of else portion

hemas

In chapter 4, we had given a list of commandline compiler option as /D that automatically creates a variable at the time of compiler running. This happens only for variables created by define preprocessor and not for any other varaibles.

Let us run the program sharp5.cs as given below

```
csc /D:vkjain sharp.cs
```

The output of this program will be displayed as shown:

Listing 13.70

vkjain

```
//sharppre3.cs
#undef vkjain class xyz5 {
   public static void Main()
{
   #if vkjain
   System.Console.WriteLine("vk jain");
   #else
   System.Console.WriteLine("hema");
   #endif
} }
```

The output of this program will be displayed as shown below:

hema

We can also use #undef also which acts exactly in reverse manner. The value of variable previously created has been destroyed due to which this program prints the message of #els portion

14

XML

14.1 XML Technology Backgrounder

Extensible Markup Language (XML) provides a means for describing structured data. Whereas HTML tags are used primarily to control display style (such as whether a word is displayed as bold or italic). XML tags are used to define the structure of data. In XML you can define an unlimited set of tags. An XML element can declare its associated data to be a retail price, a sales tax, a book title, the amount of precipitation, or any other desired data element. As XML tags can be adopted in every departement of an organization, data from all kinds of different data stores become easier to exchange and manipulate.

XML is a subset of SGML that is optimized for delivery over the Web; it is defined by the World Wide Web Consortium (W3C), ensuring that the XML standard will be uniform and independent of applications or vendors.

14.2 Uses of XML

XML, which provides a data standard that can encode the content, semantics, and schemata for a wide variety of cases ranging from simple to complex, can be used to mark up the following:

1. An ordinary document.
2. A structured record, such as an appointment record or purchase order.
3. An object with data and methods, such as the persistent format of an object or ActiveX control.
4. A data record, such as the result set of a query.
5. Meta-content about a Web site, such as Channel Definition Format (CDF).
6. Graphical presentation, such as an application's user interface.
7. Links between information and people on the Web.
8. XML is used by ADO.NET to persist and transmit data in order to facilitate the exchange of DataSets between components.

14.3 Advantages of XML

XML is simple, platform independent and is a widely adopted standard. The power of XML is that it separates the user interface from the structured data. This separation of data from presentation enables the integration of data from diverse sources. Customer information, purchase orders, research results, bill payments, medical records, catalog data etc. can be converted to XML.

14.4 Sample XML Data

If you are familiar with HTML, you can quickly learn to create XML documents.

In the following example, XML is used to describe a purchase order containing information about customers, orders, and order details:

Listing 14.1

```
<CustomerOrderDetails> <customers>
        <CustomerID>ARBORSHOE</CustomerID>
        <CompanyName>Arbor Shoes</CompanyName>
        <ContactName>Amy Anderson</ContactName>
        <ContactTitle>Sales Representative</ContactTitle>
        <Address>4567 Main St.</Address>
        <City>Buffalo</City>
        <PostalCode>98052</PostalCode>
        <Country>USA</Country>
        <Phone>555-0100</Phone>
        <Fax>555-0101</Fax>
        <orders>
            <OrderID>10643</OrderID>
            <CustomerID>ARBORSHOE</CustomerID>
            <EmployeeID>6</EmployeeID>
            <OrderDate>1997-08-25 07:00:00</OrderDate>
            <RequiredDate>1997-09-22 07:00:00</RequiredDate>
            <ShippedDate>1997-09-02 07:00:00</ShippedDate>
            <ShipVia>1</ShipVia>
            <Freight>29.46</Freight>
            <ShipName>Arbor Shoes</ShipName>
            <ShipAddress>4567 Main St.</ShipAddress>
            <ShipCity>Buffalo</ShipCity>
            <ShipPostalCode>98052</ShipPostalCode>
            <ShipCountry>USA</ShipCountry>
            <orderdetails>
                <OrderID>10643</OrderID>
                <ProductID>28</ProductID>
                <UnitPrice>6000</UnitPrice>
                <Quantity>15</Quantity>
                <Discount>0.25</Discount>
```

```
            </orderdetails>
        </orders>
    </customers>
</CustomerOrderDetails>
```

You can display this data in many different ways, or hand it off to other applications for further processing.

14.5 Elements and Attributes

XML schemas are built using the basic building blocks of XML documents: XML elements and attributes. For details, see Elements and Attributes in XSD.

14.6 XML Schemas

XML schemas are the rules that define how elements and attributes structured to form XML documents. For details, see Introduction to XML Schemas and XSD.

14.7 Style Sheets

You can also apply a style sheet to display XML data in a browser. You can use cascading style sheets (CSS) or XSL (eXtensible Stylesheet Language).

To create an XSD XML schema, you define elements and attributes. These can be typed with either the existing XML types defined by the World Wide Web Consortium (W3C) or by using simple or complex types you defined previously in the schema. You can define the structure of the elements and attributes in the document using XML schemas. Visual Studio uses the XSD standard for XML schemas.

14.8 XML Elements

XML elements are divided into two categories: simple and complex. Complex elements contain sub-elements and/or carry attributes, while simple elements contain only text (strings, dates, etc.). A complex element can be mapped to relational tables, with the simple elements defining the columns of the table. Simple elements can also be used to add content to a complex type.

The following is an example of a complex element, named 'Item', in an XML schema.

Listing 14.2

```
<element name="Item"> <complexType>
        <all>
        <element name="Quantity" type="integer" />
        <element name="Price" type="string" />
        <element name="ProductID" type="integer" />
        </all>
    </complexType>
</element>
```

Notice how the complexType tag includes additional simple elements defining Quantity, Price, and ProductID columns. In the XML Designer, element items that are dropped onto the design surface from the Toolbox are complex by default.

14.9 XML Attributes

An element can contain zero, one, or more than one attributes.The following XML defines a new attribute, names it 'City', and specifes that it must be a string.

```
<attribute name="city" type="string" />
```

Each attribute of an element can be specified only once, but in any order. For details, see attribute Element.

Attributes can be added to the XML Designer either by dropping them from the Toolbox or by right-clicking the design surface and choosing 'Add'.

Do we need an Element or an Attribute? In most cases, when used with DataSet schemas, elements and attributes are equivalent. The following list describes the key differences from a schema perspective:

1. A schema can define whether the ordering of elements is significant, but attributes can occur in any order.
2. Elements can be nested with the <choice> tag, which means that one and only one of the listed elements can appear.
3. Elements can occur more than once, but attributes cannot (columns can not recur in a dataset).

14.10 Introduction to XML Schemas and XSD

Schemas are definitions: A database schema is a description of the tables, columns, and properties of a database. Just as a database schema defines the structure of a database, an XML schema defines the structure of an XML document so that it can be shared between applications and organizations. For example, a purchase order that follows an accepted purchase-order schema can be validated with that schema before it is sent from the buyer and before the seller uses it. In Visual Studio, the XML Designer is the tool for creating and editing XML schemas and XML data files. The designer provides both visual and code representations of the schema. It can be accessed by adding either an XML Schema or DataSet item to the project or by editing an existing .xml or .xsd file.

XML schemas are used to define XML data sources. These data sources might be files, streams of bytes passing between remote applications, or other types of data sources. Schemas specify rules that define both the content and structure of the data source. Content defines which XML

elements or attributes are valid, whereas structure indicates when specific child elements are valid and whether elements can reoccur. A schema can be thought of as a class definition, and a particular XML file that conforms to that schema can be thought of as an instance of that class.

There are several different standards for defining XML schemas, including DTD (Document Type Definition), XDR (XML Data Reduced) and XSD (XML Schema Definition), the new standard recommended by the World Wide Web Consortium (W3C). Visual Studio primarily targets XSD.

14.11 Advantages of XSD

1. XSD has many advantages over the older DTD approach of defining XML schemas.
2. XSD uses XML itself to define a schema, it provides namespaces, and it is extensible.
3. XSD supports reusable simple and complex types and allows you to create new types using inheritance.
4. You can use groups to control the reoccurrence of elements and attributes.
5. You can specify uniqueness constraints and one-to-many data relationships with key and keyref statements.
6. Beta 1 Note As of this writing, the XSD standard is being finalized by the World Wide Web Consortium.

14.11.1 Sample XSD

In the sample schema below, the first line specifies the version of XML that you are using. The second line names the schema and specifies the default namespace defining the tags that can be included in the schema.

The declarations shown in the schema have the following meaning:

1. **ComplexType** - specifies a type and conditions for elements of this type, and what subelements (if any) it can contain.
2. **Element** - declares that a previously defined element type can appear within the scope of the elementType element.
3. **Attribute** - declares that a previously defined attributeType can appear within the scope of the elementType element.

Listing 14.3

```
<?xml version="1.0" encoding="utf-8"?>
<schema targetNamespace="http://tempuri.org/XMLSchema1.xsd"
xmlns="http://www.w3.org/1999/XMLSchema">
<complexType name="addressType"> <sequence>
      <element name="street1" type="string"/>
      <element name="street2" type="string"/>
      <element name="city" type="string"/>
      <element name="state" type="string"/>
      <element name="zip" type="integer"/>
</sequence>
</complexType>
```

```
<element name="purchaseOrder"> <complexType>
          <all>
         <element name="shipTo" type="addressType" />
         <element name="billTo" type="addressType" />
         <element name="shipDate" type="date" />
         <element name="item" maxOccurs="unbounded">
             <complexType>
                <all>
                <element name="productNumber" type="int"/>
                <element name="unitPrice" type="decimal"/>
                <element name="quantity" type="int"/>
                </all>
             </complexType>
         </element>
          </all>
    </complexType>
</element>
</schema>
```

14.11.2 XSD Schemas and DataSets

It is possible to generate an ADO.NET typed DataSet class from an XSD Schema. After generating the DataSet, you can create a connection to the database, allowing you to move data back and forth between the DataSet and the database. For details, see Creating ADO.NET Typed DataSets from Schemas and XML Designer and ADO.NET Design Tools.

The XML Designer generates the typed DataSet class. The class inherits from the ADO.NET DataSet and adds properties that allow direct access to the tables and columns specified by the schema.

In Visual Studio, the native serialization format for data is XML while the native serialization form for schemas is XSD.

14.12 XML Examples

C# provides a mechanism for developers to document their code using XML. In source code files, lines that begin with /// and that precede a user-defined type such as a class, delegate, or interface; member such as a field, event, property, or method; or a namespace declaration can be processed as comments and placed in a file.

The following sample provides a basic overview of a type that has been documented.

Listing 14.4

```
000: // XMLDoc\XMLsample.cs
001: using System;
002:
003: /// <summary>
004: /// Class level summary documentation goes here.</summary>
005: /// <remarks>
```

```
006: /// Longer comments can be associated with a type or member
007: /// through the remarks tag</remarks>
008: public class SomeClass
009: {
010:     /// <summary>
011:     /// Store for the name property</summary>
012:     private string myName = null;
013:
014:      public SomeClass()
015:      {
016:          //
017:          // TODO: Add Constructor Logic here
018:          //
019:      }
020:
021:     /// <summary>
022:     /// Name property </summary>
023:     /// <value>
024:     /// A value tag is used to describe the property value</value>
025:     public string Name
026:     {
027:        get
028:        {
029:           if ( myName == null )
030:           {
031:              throw new Exception("Name is null");
032:           }
033:
034:           return myName;
035:        }
036:     }
037:
038:     /// <summary>
039:     /// Description for SomeMethod.</summary>
040:      /// <param name="s"> Parameter description for s goes here</
param>
041:     /// <seealso cref="String">
042:    /// You can use the cref attribute on any tag to reference a type
or member
043:     /// and the compiler will check that the reference exists. </
seealso>
044:     public void SomeMethod(string s)
045:     {
046:     }
047:
```

```
048:     /// <summary>
049:     /// Some other method. </summary>
050:     /// <returns>
051:      /// Return results are described through the returns tag.</
returns>
052:     /// <seealso cref="SomeMethod(string)">
053:     /// Notice the use of the cref attribute to reference a
          //specific method </seealso>
054:     public int SomeOtherMethod()
055:     {
056:        return 0;
057:     }
058:
059:     /// <summary>
060:     /// The entry point for the application.
061:     /// </summary>
062:    /// <param name="args"> A list of command line arguments</param>
063:      public static int Main(String[] args)
064:      {
065:          //
066:          // TODO: Add code to start application here
067:          //
068:
069:          return 0;
070:      }
071: }
072:
```

Save the above file as xml.cs in root directory.(Not compulsory to save it at this place.)

14.13 Compilation

To compile the sample code, type the following command line:

```
* csc \XML.cs /doc:XML.xml
```

This will create the XML file XML.xml in the root directory oc c:. This file is reproduced after explanation portion.

14.14 Explanation

XML documentation starts with ///. The processing of these comments has some restrictions:

1. The documentation must be well-formed XML. If the XML is not well-formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.
2. Developers are free to create their own set of tags. There is a recommended set of tags. Some of the recommended tags have special meanings:

- The <param> tag is used to describe parameters. If used, the compiler will verify that the parameter exists and that all parameters are described in the documentation. If the verification failed, the compiler issues a warning.
- The cref attribute can be attached to any tag to provide a reference to a code element. The compiler will verify that this code element exists. If the verification failed, the compiler issues a warning. The compiler also respects any using statements when looking for a type described in the cref attribute.
- The <summary> tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

14.15 Output

Here is the resulting XML file from the class above:

Listing 14.5

```
<?xml version="1.0" ?> - <doc> - <assembly>
<name>xml</name>
</assembly>
- <members> - <member name="T:SomeClass">
<summary>Class level summary documentation goes here.</summary>
<remarks>Longer comments can be associated with a type or
member through the remarks tag</remarks>
</member>
- <member name="F:SomeClass.myName">
<summary>Store for the name property</summary>
</member>
- <member name="M:SomeClass.SomeMethod(System.String)">
<summary>Description for SomeMethod.</summary>
<param name="s">Parameter description for s goes here</param>
<seealso cref="T:System.String">You can use the cref attribute
on any tag to reference a type or member and the compiler will check
that the reference exists.</seealso>
</member>
- <member name="M:SomeClass.SomeOtherMethod">
<summary>Some other method.</summary>
<returns>Return results are described through the returns
tag.</returns>
<seealso cref="M:SomeClass.SomeMethod(System.String)">Notice
the use of the cref attribute to reference a specific method</seealso>
</member>
- <member name="M:SomeClass.Main(System.String[])">
<summary>The entry point for the application.</summary>
<param name="args">A list of command line arguments</param>
</member>
- <member name="P:SomeClass.Name">
<summary>Name property</summary>
```

```
    <value>A value tag is used to describe the property
value</value>
    </member>
    </members>
    </doc>
```

Note : The XML file does not provide full information about the type and members (for example, it does not contain any type information). To get full information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

14.16 Processing the XML File

The compiler generates an ID string for each construct in your code that is tagged to generate documentation. (For information on how to tag your code, see Tags for Documentation Comments.) The ID string uniquely identifies the construct. Programs that process the XML file can use the ID string to identify the corresponding .NET metadata/reflection item that the documentation applies to.

The XML file is not a hierarchical representation of your code, it is a flat list with a generated ID for each element.

The compiler observes the following rules when it generates the ID strings:

1. No whitespace is placed in the string.

   The first part of the ID string identifies the kind of member being identified, via a single character followed by a colon. The following member types are used:

Character	Description
N	namespace
T	type: class, interface, struct, enum, delegate
F	field
P	property (including indexers or other indexed properties)
M	method (including such special methods as constructors, operators, and so on)
E	event ! error string; the rest of the string provides information about the error. The C# compiler generates error information for links that cannot be resolved.

The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by the hash-sign ('#'). It is assumed that no item has an hash-sign directly in its name. For example, the fully qualified name of the String constructor would be 'System.String.#ctor'. For properties and methods, if there are arguments to the method, the argument list enclosed in parentheses follows. If there are no arguments, no parentheses are present. The arguments are separated by commas. The encoding of each argument follows directly how it is encoded in a .NET signature:

2. Base types. Regular types (ELEMENT_TYPE_CLASS or ELEMENT_TYPE_VALUETYPE) are represented as the fully qualified name of the type.
3. Intrinsic types (for example, ELEMENT_TYPE_I4, ELEMENT_TYPE_OBJECT, ELEMENT_TYPE_STRING, ELEMENT_TYPE_TYPEDBYREF) are represented as the fully qualified name of the corresponding full type.For example,System.Int32 or System.TypedReference.
4. ELEMENT_TYPE_VOID is represented as '=VOID'.
5. ELEMENT_TYPE_PTR is represented as a '*' following the modified type. ELEMENT_TYPE_BYREF is represented as a '@' following the modified type.
6. ELEMENT_TYPE_PINNED is represented as a '^' following the modified type. The C# compiler never generates this.
7. ELEMENT_TYPE_CMOD_REQ is represented as a '|' and the fully qualified name of the modifier class, following the modified type. The C# compiler never generates this.
8. ELEMENT_TYPE_CMOD_OPT is represented as a '!' and the fully qualified name of the modifier class, following the modified type.
9. ELEMENT_TYPE_SZARRAY is represented as '[]' following the element type of the array.
10. ELEMENT_TYPE_GENERICARRAY is represented as '[?]' following the element type of the array. The C# compiler never generates this.
11. ELEMENT_TYPE_ARRAY is represented as [lowerbound:size,lowerbound:size] where the number of commas is the rank -1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is simply omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. For example, a 2-dimensional array with 1 as the lower bounds and unspecified sizes is [1:,1:].
12. ELEMENT_TYPE_FNPTR is represented as '=FUNC:type(signature)', where type is the return type, and signature is the arguments of the method. If there are no arguments, the parentheses are omitted. The C# compiler never generates this.

The following signature components are not represented because they are never used for differentiating overloaded methods:

- calling convention
- return type
- ELEMENT_TYPE_SENTINEL

For conversion operators only (op_Implicit and op_Explicit), the return value of the method is encoded as a '~' followed by the return type, as encoded above.

The following example show how the ID strings for a class and its members would be generated:

Listing 14.6

```
namespace N  // "N:N"
{
   class X   // "T:N.X"
   {
         X();   // "M:N.X.#ctor"
```

```
        static X(); // "M:N.X.#cctor"  a class constructor
        X(int);  // "M:N.X.#ctor(System.Int32)"
        ~X();   // "M:N.X.Finalize" destructor's representation in
metadata
        string q;  // "F:N.X.q"
        const double PI = 3.14;  // "F:N.X.PI"
        int f();  // "M:N.X.f"
        int bb(string s, ref int y, void * z);
        // "M:N.X.bb(System.String,System.Int32@,=VOID*)"
        int gg(short[] array1, int[,] array);
        // "M:N.X.gg(System.Int16[], System.Int32[0:,0:])"
        X operator+(X, X); // "M:N.X.op_Addition(N.X,N.X)"
        int prop {get; set;} // "P:N.X.prop"
        event MyDelegate d; // "E:N.X.d"
        int this[string s]; // "P:N.X.Item(System.String)"
        class Nested{} // "T:N.X.Nested"
        delegate void D(int i); // "T:N.X.D"
        explicit operator int(X x);
// "M:N.X.op_Explicit(N.X)~System.Int32" } }
```

14.17 Tags for Documentation Comments

The C# compiler will process documentation comments in your code to an XML file. Processing the XML file to create documentation is a detail that needs to be implemented at your site.

Tags are processed on code constructs such as types and type members.

Note : Tags are not processed on namespaces. The compiler will process any tag that is valid XML. The following tags provide commonly used functionality in user documentation:

- **<c> <param>1 <see>1**
- **<code> <paramref>1 <seealso>1**
- **<example> <permission>1 <summary>**
- **<exception>1 <remarks> <value>**
- **<list> <returns>**

15

Building Projects

15.1 Projects in Visual Basic and C#

A project is made up of separate components that are stored as individual files in a solution. A simple project may consist of a form or HTML document, source code files and a project file. More complex projects consist of these items plus database scripts or stored procedures, a reference to an existing Web service, a project functions like a container from which you can assemble components and deploy a finished application. The output of a project is usually an executable program (.exe), a dynamic link library (.dll) file or a module, among others.

All projects are contained within a solution and every project contains a unique project file. A project file is a list of the files contained in a project. It is also used to store and track other information about those files. This file is updated every time you save the project.

All Visual Studio.NET development products provide a number of pre-defined project templates. A project template creates a new project in your solution and populates it with the files you will need to develop a project of that type.

There are two main categories of projects in Visual Basic and C#: local projects and Web projects. The main distinction between the two categories is in where the project files are located. For a local project, files are located on your development computer; for a Web project the files are located on a Web server.

Local project types include projects for building Windows applications (.exe files), components (.dll files), console applications, and Windows services. Middle-tier components are considered to be local since they are developed on the local computer, regardless of where they will be deployed.

Web project types include Web applications and Web Services. A Web application can include dynamic elements such as Web Forms or Web Services, or it may be simply a set of static Web pages.

When you create a new project, icons in the New Project dialog box and Add Project dialog box represent the available project types and their templates. The project template that is associated with the project icon you choose determines the output type and other options available for that project. The following are the types of projects available for Visual Basic and C#.

Windows Application :A project for creating an application with a Windows Form-based user interface.

ClassLibrary:A project for creating classes to use in other applications .

Windows Control Library :A project for creating controls to use in Windows-based applications.

WebApplication:A project for creating an application with static or dynamic Web pages as a user interface. It resides on a Web server.

WebService:A project for creating Web Services to use from other applications.

WebControlLibrary:A project for creating controls to use in Web applications.

Console Application :A project for creating a command-line application.

WindowsService:A project for creating services for Windows.

Import Folder Wizard (Visual Basic only) : A wizard for creating a project in an existing folder.

EmptyProject:An empty project for creating a local application.

EmptyWebProject:An empty project for creating a Web application.

15.2 Project Items

Project items refer to the types of files added to a project. The project items available to you depend on whether you are in a local project or a Web project.

When you add a project to a solution, the project template associated with an icon in the Add New Project dialog box often provides you with more than one default item type appropriate for your project. However, when you use the Add New Item dialog box to add an individual project item, Visual Studio provides you with a single empty starter file of a particular type, for example, a blank Windows Form, a blank class, or a blank user control, and Visual Studio creates a file for it within the project in Solution Explorer. For a description of each item type listed in the Add New Item dialog box, see the tables later in this topic.

15.3 Adding Project Items

The Add New Item dialog box is accessed through the Project menu. The Categories pane displays one of the two possible nodes, Local Project Items or Web Project Items, depending on whether you are in a local project or a Web project. When you select a folder in the Categories pane, all item types under that folder are displayed in the Templates pane. You can select a subfolder in the Categories pane to display a subcategory of items in the Templates pane.

In the Templates pane when you select one of the icons representing a project item, a brief text description of the item appears in a panel at the bottom of the Add New Item dialog box.

Note that for some project items, the file extension differs depending on whether the item occurs in a Visual Basic project (indicated by .vb) or in a C# project (indicated by .cs). For other project items, the file extension does not distinguish between a Visual Basic project and a C# project; that is, there is only one possible file extension for the project type.

15.4 Local Project Items

Project item:Windows Form

File extension:.vb or .cs

Purpose of the project: A basic Windows form that you use for local applications. It has a graphical design surface. InheritedForm.vbor.csA Windows form that you derive from another form by using visual inheritance. When you inherit from another form, you receive this form as your initial form. You can then visually add to it or change its contents. UserControl.vbor.csA control (a visual element) that you can place on a Windows form. It has a visual designer. Inherited User Control .vb or .cs Similar to a User Control except that it is derived from another existing user control. CustomControl.vbor.csA Windows control that is not visually designed, i.e.it does not have a graphical design surface. You must create the control by writing code. Class.vbor.csA code file that initially contains a single empty class declaration. Module (Visual Basic only) .vb A code file that initially contains a single file for storing functions. CodeFile.vbor.csAn empty Visual Basic or C# code file; it contains no code. WindowsService.vbA file used to create a Windows service. It contains a visual designer and a class containing methods to create the service. InstallerClass.vbor.csA file used to create a custom setup action for deployment, for example, a special procedure you want to run during setup to install a message queue. CrystalReport.rptA file for a Crystal Report (a third party database reporting tool).When you open the file, it opens in the Crystal Report Designer. Dataset.xsdAn XSD schema that generates classes for the dataset that you can use to access the data programmatically. XSDSchema.xsdAn XSD schema file without the generated classes. XMLFile.xmlAn XML document file. XSLTFile.xsltA file used to transform XML and XSD documents. HTMLPage.htmA typical Web page. Frameset.htmAn HTML frameset, an outer-level page that allows you to host multiple HTML pages in different frames. Stylesheet.cssA cascading style sheet file (CSS) that is used with HTML Web pages to define styles. TextFile.txtAn empty text file. VBScriptFile.vbsAn empty Visual Basic Scripting Edition (VBScript) code file. JScriptFile.jsAn empty JScript.NET code file. WindowsScriptHost.wsfAn empty code file used for Windows scripting.

15.5 Web Project Items

Project Item	File Extension	Purpose of the Project Item:
WebForm	.aspx	A form used in a Web application, analogous to a Windows form in a local application. It provides a visual designer, that is, a graphical design surface.
HTMLPage	.htm	Same as for Local Project Items.
Frameset	.htm	Same as for Local Project Items.
Stylesheet	.css	Same as for Local Project Items.
ActiveServerPage	.aspAn	ASP page, used to define both the HTML formatting and the server-side script.
WebService	.asmx	A class that exposes methods via HTTP requests. It provides a graphical design surface.

contd...

Project Item	File Extension	Purpose of the Project Item:
Class	.vb or .cs	Same as for Local Project Items. Module (Visual Basic only) .vb Same as for Local Project Items.
CodeFile	.vbor.cs	Same as for Local Project Items.
InstallerClass	.vbor.cs	Same as for Local Project Items.
Dataset	.xsd	Same as for Local Project Items.
CrystalReport	.rpt	Same as for Local Project Items.
XSDSchema	.xsd	Same as for Local Project Items.
XMLFile	.xml	Same as for Local Project Items.
XSLTFile	.xslt	Same as for Local Project Items.
Dynamic Discovery Document	.disco	Sometimes called a disco file— this provides a means to enumerate all Web Services and all schemas in a Web project.
WebConfig-urationFile	.webASP.NET	uses this file to configure the Web settings for a Web project. The file has a name of config.web which you cannot change.
Global Application Class	.asax	sometimes called an asax file—the file allows you to write code to handle global ASP.NET application-level events such as Session_OnStart and Application_OnStart. The file has a name of global.asax which you cannot change.
TextFile	.txt	Same as for Local Project Items.
WebCustomControl	.vbor.cs	Same as for Local Project Items.
VBScriptFile	.vbs	Same as for Local Project Items.
JScriptFile	.js	Same as for Local Project Items.
WindowsScriptHost	.wsf	Same as for Local Project Items.

15.6 Web Projects

Web projects are developed on a Web server as against being created on a local computer. You can use these projects to create Web sites with Web Forms, Web Services, HTML pages, and so forth.

15.6.1 Web Project File Storage

When you create or open a Web project, you specify a URL for the location of the project. The files in the project are stored on the Web server at the location that you specify.

The project also keeps temporary copies of your files in a cache location. These are used by the project system while you are working offline from the Web server. For details, see *Taking Web Projects Offline*.

15.6.2 Web Project Build Model

When you are developing a Web project, you can write code in project files. Unlike the script code in Active Server Pages (ASP) that is interpreted at run time, this code is compiled before it is run on the Web server. Compiled code provides run-time performance advantages. When a Web project is built, the compiled output is deployed to the Web server.

You can view a Web page in Visual Studio by selecting its node in Solution Explorer and choosing View in Browser on the File menu. To ensure that this view is accurate, after making changes, you must build the project before choosing View in Browser.

15.6.3 Server Requirements for Web Projects

In order to develop, debug, and run Web projects you must install on your server: Internet Information Server (version 4.0 or higher), FrontPage Server Extensions, the .NET Framework Common Language Runtime, and ASP.NET.

15.6.4 Web Access Methods

A Web access method is the method of accessing the files you place on a Web server when you are developing a Web project Microsoft Visual Studio.NET allows two different types of access to Web projects:

- FrontPage access
- File Share access.

FrontPage Server Extensions are used on a Web server to support Web pages designed in clients like Microsoft FrontPage. When you use the FrontPage access method, Visual Studio accesses your project files on the Web server through the Server Extensions. Using the File Share access method, you can directly access the project files on disk without needing to go through the Server Extensions.

15.6.5 Choosing the Best Access Method

There are two primary considerations in determining the appropriate Web access method.

The first depends on where your Web server is located. If you do not have local access to a shared folder on the Web server, then you cannot use the File Share access method. FrontPage Server Extensions use the HTTP protocol, so the FrontPage access method can be used to develop Web projects over the Internet. But if you are developing a Web project on a local intranet server, then using File Share access eliminates the need for FrontPage Server Extensions on the local server.

The second consideration depends on the source control model you prefer. Once a Web project is placed under source control, development clients can only connect to the site using one type of access method. Using FrontPage source control integration with Visual SourceSafe controls, any changes can be made to the Web project files through the Server Extensions. In this model, only one copy of the Web project exists, and all developers work in this single project. Each developer checks out a particular file in exclusive mode; only that developer can edit that particular file. Once a Web project is placed under the FrontPage source control, the project can only be opened using FrontPage access.

A File Share accessed Web project added to source control behaves similarly to local projects under source control. Developers check out copies of the Web project files from source control, and then check their changes back into source control. Once a project is placed under File Share source control, the project files can only be checked out using File Share access.

15.6.6 When to Specify the Web Access Method

The default Web access method for Web projects created in Visual Studio.NET is File Share. However, before you create a new Web project, you can set the preferred Web access method using the Options command on the Tools menu. For more information on starting a new project, see *Creating a Web Project*.

After a Web project is created, the Web access method is stored and used the next time the project is opened. If there is a problem when creating or opening a Web project, you can switch to access methods by using an error dialog box. Once open, you can change its access method in the Project Property Pages dialog box, as long as the Web project is not under source control. Visual Studio will use the new access method the next time the Web project is opened.

15.6.7 Adding Projects to Source Control

Even though you can choose your own project templates in Visual Studio.NET, the project model upon which those templates are based cannot be specified. You can choose to implement one project model in preference to another only by selecting a particular project template. In fact, you will encounter no reference to these models or the terms used to describe them while working in the Visual Studio environment.

15.7 Project Behavioral Model

A set of rules govern how a user interacts with a project and how projects behave in relation to one another. Project behavioral models represent design patterns created to ensure a consistent and reliable user experience in the integrated development environment (IDE).

15.7.1 Project Behavioral Models, Classified and Differentiated

If you have created a project in Visual Studio.NET, you have already encountered one of the following project behavioral models:

- Storage-based;
- Reference-based; or
- Hybrid i.e. a combination of the two—a hybrid model.

The following descriptions provide a brief introduction to each model.

(1) Storage-based Project

They are marked by the following characteristics.

- Project items must all reside in a root project directory or a subdirectory of that root directory.
- Contains a project file but does not use that file as its only method of defining the list of project items and their respective locations.
- All items in the project directory will be built.
- Cannot contain references to items in other projects.
- Web projects are common examples.

(2) Reference-based Project

They have the following characteristics:

- Items can be stored anywhere.
- Uses a project file to list and track files contained in the project, source code control information and other metadata.
- Only the project items listed in the project file as included will be built.
- Can contain references to items in other projects. Visual C++ projects are common examples.

(3) Hybrid Project

They have the following characteristics:

- They exhibit behaviors characteristic of both Storage-based and Reference-Based project behavioral models.
- Items can be stored anywhere.
- Uses a project file to list and track files contained in the project, source code control information and other metadata.
- Can usually contain references or links to files in other projects. Visual Basic.NET and C#.NET are common examples.

15.7.2 Project Type Behavioral Model

Visual C++.NET Reference-based Web Storage-based C#.NET Hybrid (primarily storage-based) Visual Basic.NET Hybrid (primarily storage-based) Visual Fox Pro Hybrid (primarily storage-based)

Note that although they are not projects, Visual Studio.NET solutions behave according to the rules of a purely reference-based project. For example, the items in the Solution Items folder of Solution Explorer behave like items in a Visual C++.NET project. User interaction with projects occurs primarily during the creation and maintenance of project items. A project item inherits the behaviors of its parent project. Thus, the effect of project behavioral models is primarily evidenced in the behavior of project items.

Similarly, the majority of project to project interaction centers around the creation and maintenance of project items. Most users manage their projects and the items they contain in Solution Explorer. Therefore, the tangible benefits of consistent project behavior is most important.

> **Note : Visual C++.NET and Web projects subscribe to pure project behavioral models of different types. For this reason, the interaction between two such projects —and your interaction with them —is likely to manifest the most number of differences. Command Availability and Project Behavioral Models**

As previously mentioned, the relationship between projects and project behavioral models is pre-defined. Nevertheless, your awareness of the application design and project maintenance considerations attending the existence of project behavioral models is vital to your understanding of command availability in the IDE. When you select a Web project item e.g.the commands by which you can manipulate that item differ from the commands available for manipulating C#.NET project items.

The behavioral model affects how projects behave in the context of the Visual Studio environment in several key scenarios. These scenarios include: moving and copying items in Solution Explorer; saving files as new file types; and deleting project items.

15.7.3 Moving and Copying Items

The outcome of drag and drop or cut and paste operations in Solution Explorer depends upon the project behavioral model context of both originating and destination nodes. The following matrices summarize the drag and drop operation that is performed on a project item based on its source project, target location and modifier keys pressed.

From reference-based To reference-based:

- Modifier Effect
- none Move
- CTRL Copy
- SHIFT Move
- CTRL+SHIFT Copy

From reference-based To storage-based:

- Modifier Effect
- none Copy
- CTRL Copy
- SHIFT No drop
- CTRL+SHIFT Create reference

From storage-based To reference-based:

- Modifier Effect
- none Create reference
- CTRL Create reference
- SHIFT No drop
- CTRL+SHIFT No copy, no move

From storage-based To storage-based:

- Modifier Effect
- none Move
- CTRL Copy
- SHIFT Move
- CTRL+SHIFT Create reference

15.8 Saving Project Items

The effects of using the Save As command for project items depends on the project behavior model to which its project subscribes.

Reference-based projects

Save As creates a new file outside the project hierarchy. The file appears in Miscellaneous Files, if the Show Miscellaneous Files in Solution Explorer option is enabled,then see Documents, Environment, Options Dialog Box for more information.

Storage-based projects

Save As replaces the current file. The old file is removed and the new file created in its place.

Hybrid project

Save As replaces the current file if the new copy is saved to the same directory. Otherwise, Save As replaces the current file as a link.

15.9 Removing Project Items

The project behavioral models determine the method or methods by which items can be removed from a project in a given context.

Remove: This command is available in hybrid and storage-based projects. When you remove a file from your project the file is disassociated from the project and disappears from Solution Explorer. Removed files are not permanently removed from storage. Remove is not available for use in the context of Reference-based projects such as Visual C++.NET.

Exclude: This command is available for practically all project types. With Exclude, you can temporarily remove an item from the project build and subsequently re-introduce it with Show All Files. For the steps needed to exclude an item from your project builds, see Adding and Removing Project Items. Adding a file to a project directory in Windows Explorer does not automatically add that file to the project; the file will appear in Solution Explorer in the Show All Files view but is not part of the project until you select Include in Project from the Project menu.

Delete: This command applies exclusively to hybrid and storage-based projects. Deleted items are permanently removed from file storage and disassociated from the project. Reference-based projects, as well as non-project solution items, do not support the Delete command.

15.10 Referencing

In order to write code against an external component, your project must first contain a reference to that component. A reference can be made to three types of components: .NET assemblies, COM automation servers and controls, and other projects in the same solution which expose components.

To add a reference:

1. In Solution Explorer, select the project node to which you want to add the reference.
2. From the Project menu, choose Add Reference.
3. In the Add Reference dialog box, click either the .NET Framework, COM, or Projects tab, depending on the type of component you want to reference.
4. The .NET Framework tab lists all of the assemblies in the .NET Framework directory. The COM tab lists all of the registered type libraries on the local computer. The Projects tab lists all the projects in your solution.
5. In the top pane, select the component you want to reference, and then click the Select button. If the component you are looking for is not in the list, you may locate it on disk using the Browse button.
6. The component you referenced appears in the SelectedComponents window at the bottom of the Add Reference dialog box.
7. You can continue selecting more components, or you can click OK to finish adding the references to your project.
8. The components you referenced are added to the project, and they appear in Solution Explorer. Under the References node within the project in Solution Explorer is a node with the name of the component you just referenced. At this point you can use any of the public code (for example, a Public method) contained in the referenced component in your project.

Note : Toolbox controls do not need to be referenced, because the references are added automatically to a project when you drag the control from the Toolbox to a designer. Adding a reference to an item that is already referenced has no effect on your project.

9. Select the reference in Solution Explorer, and choose Remove from the Edit menu.

15.11 Working with Projects

Broken references can occur in several situations:

- The project's reference path is incorrect or incomplete (The file being referenced has been deleted);
- The file being referenced has been renamed;
- The file being referenced is for some other reason inaccessible to your project;or
- The Reference Path is Incorrect.

If projects are shared on different computers, some references might break if a component is located in a different directory on each computer. References are stored under the name of the component file (for example, MyComponent.dll). When a reference is added to a project, the folder location of the component file (for example, C:\MyComponents\) is appended to the Reference Path project property.

When the project is opened, it attempts to locate these referenced component files by looking in the directories on the Reference Path. If the project is opened on a computer that stores the component in a different directory, such as D:\MyComponents, the reference cannot be found and an error appears in the Task List.

To solve this problem, you can delete the broken reference and then replace it using the Add Reference dialog box. Alternatively, you may use the Reference Path item in the project's property pages and modify the folders in the list to point at the correct locations. The Reference Path property is persisted for each user on each computer, so modifying your reference path does not affect other current users of the project.

To fix a broken project reference by correcting the reference path

1. In Solution Explorer, right-click your project node.
2. On the shortcut menu, choose 'Properties'.
3. The Property Pages dialog box for your project appears.
4. Select the appropriate property page in the dialog box.
5. If you are using Visual Basic, select 'Reference Path' under the Common Properties folder in the left pane. In the 'Reference Path' pane, type the path in the Folder field, and then click the 'Add Folder button'.
6. If you are using C#, select 'Build under the Common Properties' folder in the left pane. In the Reference Path field, type the path to the folder that contains the item you want to reference, and then click OK.

Note: You can also use the ellipsis button in either Visual Basic or C# to browse to the reference path you want. Note You can add multiple directories by typing a semicolon between each entry.

Referenced File Has Been Deleted

It is possible that the file being referenced has been deleted and no longer exists on the drive.

To fix a broken project reference for a file that no longer exists on your drive, delete the reference.

Referenced File Has Been Renamed

It is possible that the file being referenced has been renamed.

To fix a broken project reference for a file that has been renamed delete the reference, and then add a reference to the new name.For more information, see Adding and Removing References.

Referenced File Is Inaccessible For Some Other Reason

There can be a variety of causes for broken references: a failed network connection or a failed authentication. Each cause might have a unique means of recovery. However, deleting the reference and fixing the code which used it is always an option. For more information, see Adding and Removing References.

You manage project Web references by renaming or updating them to suit your needs or to incorporate modifications made by others.

15.12 Renaming a Project Web Reference

Sometimes you need to change the programmatic Namespaces of Web references. For example, if you do not want to use the server name to identify a Web reference class in code, you can change the Web reference name.

To rename a Web reference in a project:

1. In Solution Explorer, under your project's Web References folder, select the node for the reference you want to rename.
2. On the File menu, choose 'Rename'.
3. You can also right-click the node and choose Rename. A box appears around the selected reference.
4. Type the new name that you want, and press ENTER.

> **Note: If you have used the old name elsewhere in your project code, remember to change each occurrence to use the new name.**

15.13 Updating a Project Web Reference

If your application contains a Web reference to a Web Service that is modified on the server, you may need to update the reference in your project. One occasion when you may need to update a Web reference is when new methods are added to an existing Web Service on a server.

To update a project Web reference:

1. In Solution Explorer, under your project's Web References folder, select the node for your Web reference.
2. Right-click the node to display the shortcut menu.
3. Choose 'Update Web Reference'.

The new files for the Web Service are downloaded into your project. Information for the Web Service is updated within your project.

15.14 Adding Web Reference

You can add a Web reference to any Web-based application. Web references allow you to access and use existing Web Services on the Internet.

To add a Web reference to a project:

1. From the Project menu, choose 'Add Web Reference' for projects that support this command.
2. In the Add Web Reference dialog box, type the URL for the Web Service you want to access. Add Web Reference displays information about the discovery files, schemas and Web services available on the site you select. Use this format for the URL: http://servername/projectname/webservicename.disco
3. Parameter Value servername The name of the server on which the Web Service was deployed. Projectname The name of the folder containing the Web Service. In most cases, this corresponds to the project name of the Web Service. Webservicename.disco The name of the discovery file that was generated for the Web Service on compilation. This should take the form of the name of the class on which your Web Service is based, followed by the file extension .disco.
4. Choose 'Go to' or press Enter. In the right half of the dialog box, navigate through the available services until you find the one you want. Select that Web service from the Components found list.

   Tip: The right half of this dialog box displays information about the currently selected item. You can use this function to learn more about your select, or you can select the associated documentation link if one is available.

5. Choose 'Add Reference'. The system adds a folder to your current project called Web References. Within that folder, a sub-folder bearing the name of the selected Web service's URL is created. Inside this sub-folder, a strongly-typed proxy object for the selected service is created.

All project references have a CopyLocal property that determines whether the component is copied locally to the project under the '\bin' directory. References made to components within the .NET run-time directory have CopyLocal set to false by default; these components are shared components and do not change often. All other references have CopyLocal set to true by default.

In order to set CopyLocal to false, the reference must be to a shared component, and the component should not be built from a project within the current solution. The compilers or the designers may hold an exclusive file access lock on references whose CopyLocal property is set to false, so that the component cannot be changed while the project is open.

15.15 Creating New Solutions and Projects

The following process can be used to create a new project and a new solution to contain it. As a project is defined by the files it contains, a solution defined by its projects. Thus, whenever you create a project in an empty workspace, a solution is created by Visual Studio to contain it. For more conceptual information, see Solutions and Projects.

This process may be used to create relatively simple solutions and projects. After following these short steps, you will be ready to add project and solution items. If you plan to create a complex or multi-project soltuion, see *Creating Multi-Project Solutions* for more instructions.

To create a new project and a solution to contain it:

1. From the File menu, select 'Project' from the New submenu.
2. In the New Project dialog box, select a 'Project Type' from the left pane.
3. Select a 'Project Template' from the right pane.
4. Click Open.

> Tip: You can also use the New Project dialog box to add a new project to an open solution by selecting the Add to Solution option. This option only appears when a solution is open in Solution Explorer.

15.16 Solutions and Projects

Projects are the building blocks of a Visual Studio application, while a solution is the foundation.

A single solution can contain multiple projects. To manage and monitor your projects and the solution to which they belong, Visual Studio.NET introduces Solution Explorer, which displays the hierarchical structure of your solutions and the projects they contain. As in the previous versions of Visual Studio, you can manage your projects individually. However, the introduction of the solution and Solution Explorer to the Visual Studio architecture facilitates numerous tasks, especially for developers of a multi-project solution.

15.17 Projects and Solutions

Conceptually, the solution is a logical container for the projects and the solution items that can be built into an application in Visual Studio.NET 7.0. The solution is also used to manage miscellaneous files that are opened outside the context of a solution or project. Miscellaneous files are not built into an application. Solutions are managed groups of related projects_analogous to group (.vbg) or workspace (.dsw) files in Visual Basic and Visual C++, respectively.

You can create an empty solution, one that contains only projects, or one that contains only miscellaneous files. However, if a solution does not already exist in Solution Explorer when you create a new project or file, a solution is automatically generated by Visual Studio.NET to contain it.

Practically, the solution model allows you to logically organize your multi-project solutions. Solution Explorer is the user interface through which you can centrally organize and manage all of the projects and files needed to design, develop and deploy an application. The following topics provide more detailed information about solutions in Visual Studio.NET.

15.18 Solution Directory Structure

In Solution Explorer, a solution logically contains the projects, solution items and files needed to build a Visual Studio application.

Important : Depending on how it was created, the logical relationship between a solution and its components (as depicted in Solution Explorer) does not necessarily mirror the way the solution and its components are stored on disk.

When you create an application consisting of multiple projects, it is best to create a local solution directory to contain your local (non-Web) projects, solution files (.sln and .suo) and any shared Solution items. A solution directory-based structure represents the organizational structure of your solution in Solution Explorer and thus, maintains the logical relationships of components on disk. Depending on whether your application will be entirely local or might potentially contain some Web-based projects, you can create a solution directory in one of the following two ways:

1. Create a Blank Solution and subsequently add projects and files.
2. Creat the first project in a solution with the New Project dialog box and select the Create directory for Solution option.

Tip : You can more easily "share out" numerous and related, source-controlled projects by creating them under a common solution directory.

15.19 Solution Explorer

Solution Explorer is a two-way interface between you and your solution. It provides real-time information about items in the open solution and allows you to manage those items. For example, you can monitor the current status of source controlled files, drag and drop items from one project to another and display solution, project or file properties. To access Solution Explorer select Solution Explorer from the View menu or type Ctrl+Alt+L.

(1) Managing items in Solution Explorer

Many of the more common file and project maintenance-related tasks can be accomplished in Solution Explorer.

(2) Moving and Copying Items

You can move, copy or create a reference to an item in Solution Explorer by simply dragging it to another location. When you position a dragged item above its target location, a ToolTip image identifies whether the item will be moved, copied, referenced or denied access to that location.

Important: The outcome of drag and drop or cut and paste operations in Solution Explorer depends upon the project user model context of both originating and destination nodes. For a description of these project user models and a discussion of their effects on all commands in the Visual Studio.NET environment, see Storage-Based and Reference-Based Projects.

Note: Solution Explorer does not support right-click drag and drop behavior.

Opening Files: Double-clicking a file performs the same action as Open. You can change the item's default editor with the Open With dialog box. Right-click the file and choose Open With from the shortcut menu.

Multiple-selection: You can select multiple items in a single project or multiple items spanning several projects. Select multiple items when you want to perform batch operations (simultaneously opening multiple files for editing, for example) or when you want to determine or edit the properties shared by two or more solution items.

When you select multiple items, the available commands represent the interesection of commands common to all selected items.

Startup Project: The startup project is identified in Solution Explorer by its bold font. When you select Multiple Startup Project from the Startup Project, Solution Property Pages, Properties page, the solution node is designated as the startup project and consequently appears in bold. Regardless of which project or solution item has focus in Solution Explorer, the startup project is built when you press the F5 key. In multiple startup project mode, you must select Build Solution from the Build menu to build the startup project.

> Tip: To conserve space in the integrated development environment (IDE) you can autohide Solution Explorer by right-clicking its title bar and selecting Autohide.

15.20 Solution Explorer Toolbar Icons

Solution Explorer toolbar icons are described below. Only buttons specific to a selected file or other item appear. For example, when you select an .asax file, five buttons appear on the toolbar. However, when you select an .asp file only the Properties icon appears.

Show All Files: Shows all solution folders and files, including those that have been excluded from the current build and those that are normally hidden.

Why are some files hidden? Do we need to mark them as hidden in the directory? When would a user want to see hidden files?

Open: Opens the selected item. You can reset the default command on a given file type in the Open With dialog box. Right-click on the file in Solution Explorer and choose Open With from the shortcut menu.

View Code Opens the selected file for editing in the Code Editor.

Refresh
Refreshes the state of selected items in the active view.

Tip: If you are working with several developers and want to refresh your local version of files others have worked on, you can get the latest version of read-only files currently in your local application by selecting this button.

Properties

Displays properties for the selected Solution Explorer item. When a project or solution node is selected, this button invokes either the Project Property Pages dialog box or the Solution Property Pages dialog box. If a file is selected, clicking this button invokes the Properties window, if it is not already displayed.

15.21 Signal Icons in Solution Explorer

Solution Explorer provides information about the status of your files, as shown by the following signal icons:

Item Not Found

Item has been moved, renamed, or deleted from the project directory.

Checked In

Item is checked out of a source control database.

15.22 Multi-Project Solutions

A solution can contain multiple projects that you can open, close, and save together. Each project in a solution might contain multiple files or items. The types of items contained within a project vary according to the development language used to create them.

Depending on how you create it, the logical relationship between a solution and its components does not necessarily mirror the way the solution and its components are stored on disk. For more information, see Solution Directory Structure. For step-by-step instructions detailing how to establish a solution directory-based structure, see Creating Multi-Project Solutions.

When you create a multi-project solution, the first project created becomes the startup project by default. The startup project appears in bold font in Solution Explorer and is the project that runs when you click Start on the Debug menu. You can also start debugging all projects in a solution simultaneously or debug one or multiple projects in the solution by selecting the solution as startup project.

When you create a multi-project solution, you can build either a single project within the solution or multiple projects in the solution. You can also specify which solution projects you wish to exclude from builds.

When you build an entire solution, projects are built in the order they were added to the solution. Dependent projects are built after the projects on which they depend.

15.23 Solution Items

In addition to your projects and the files of which they consist, a solution also contains any project-independent files you create. There are two types of project-independent files: shared solution items and miscellaneous files.

Add new or existing solution items to Solution Explorer's Solution Items node that are integral to the development of your multi-project solutions but belong to no project in particular. A solution item is deployed with its solution and applied to source control along with projects and project files.

Since solution items are integral to and owned by the projects in a solution, saving them in the root solution directory is generally recommended. However, solution items are persisted across solution sessions and are referentially associated with solutions. In other words, the solution file (.sln) remembers and more importantly tracks, the location of its solution items. Referential association means that you can save solution items to any location. Cascading style sheets are common solution items. A single stylesheet for example, can standardize the formatting of numerous Web pages in multiple projects. When the solution is closed, the stylesheet stays with its Web pages. When the solution is re-opened, the stylesheet appears beneath the Solution Explorer Solution Items node in Solution Explorer.

15.24 Miscellaneous Files

This topic is divided into the following sections:

1. When to create a miscellaneous file.
2. When to change the default behavior.
3. When to accept the default behavior.
4. What is a miscellaneous file?

A miscellaneous file is any item that is not contained by a project or solution. Miscellaneous files are external to solutions, are not included in builds and cannot be included with a solution under source control.

Miscellaneous files are however, referentially associated with a solution. This means that when you open a solution, all the miscellaneous files that were opened when the solution was last closed will be re-opened.

When to create a miscellaneous file?

Open a new or existing miscellaneous file when:

1. You intend to consult while developing a project-based solution but are not integral to the solution's development. Common examples include development notes or instructions, database schema and code clips.
2. You want to create a stand-alone file.

How miscellaneous files affect command availability in the IDE You might have noticed that menus, toolbars and the commands they contain, appear and disappear when you activate a miscellaneous file from within Solution Explorer or from within the document window itself. When you open a Text file for example, the Text Editor toolbar appears and its commands are available. If you then open an XML Schema file, the XML Schema toolbar appears. While editing your XML Schema, the Text Editor toolbar's commands (or the toolbar itself) are unavailable. The XML Schema is the active window and as such, has current selection context.

The provision of Visual Studio documentation and the population of available commands in toolbars and menus is based on your current selection context.

15.25 Current Selection Context

The existence or non-existence of an element or entity is determined by the presence and prioritized relationships of other elements in a shared environment. When you switch between a project file and a miscellaneous file, all project-related commands disappear and only those that are directly related to the miscellaneous file appear.

15.25.1 When to change the default behavior?

If you typically create a project first and add files later AND if you usually create applications that are compiled from groups of interrelated project files, the default context tracking and visibility of miscellaneous files is probably less than ideal. For more information about hiding the Miscellaneous files node in Solution Explorer, see Documents, Environment, Options dialog box.

15.25.2 When to accept the default behavior?

If you commonly use the New File command from the File menu AND you typically work with relatively independent files outside the context of a project, the default visibility of Miscellaneous files in Solution Explorer is probably desirable. The solution file (.suo) remains sensitive to your selection context and thus filters out unnecessary commands. In an uncluttered command environment, finding the commands you need to perform a given task is consequently easier.

15.26 Solutions, User Interface Elements

A solution contains projects and their items and optionally, project-independent files of two types which share solution items and miscellaneous files. Solutions allow you to centrally organize all of the projects and files needed to design, develop, and deploy an application or component.

Solution Explorer Illustrates the relationships between the parts of Solution Explorer. AddNew Project Dialog Box. Use this dialog box to add new projects to your application, using a wide variety of projects. Add Existing Project Dialog Box Documents the elements of this dialog box and provides tips to help you manage the process adding existing projects to your solution. Add New Solution Item Dialog Box Lets you choose a file type and add a stand-alone item to the current solution. Add Existing Solution Item Dialog Box Describes user interface elements and provides links to topics that discuss why you might want to add existing solution items, how to add them, and how they relate to projects. Open Solution Dialog Box Provides in-depth

documentation for the features and functionality of the Open Solution dialog box. It also gives you links to more conceptually-oriented topics that describe solutions in greater detail. Project Dependencies A brief description of this dialog box with links to more detailed topics describing dependencies and their relation to references. Startup Project, Solution Properties Dialog Box Outlines the key user interface features of the Solution Property Page: Startup Project.This topic describes how to access the page and points to related conceptual topics. Signing Options, Solution Properties Dialog Box Describes the functional features of this dialog box and provides links to related documentation. Configuration Manager Dialog Box In-depth information about the dialog box that allows you to select which projects to build or deploy with your solution and change each project's configuration and platform. New Solution Configuration Dialog Box Describes the features of this dialog box , as well as under what conditions you might use them.

15.27 Add Existing Project Dialog Box

Use this dialog box to select existing projects and add them to your solution. You can add an existing project from local or network directories and Web sites. Access the 'Add Existing Project' dialog box by selecting 'Add from the File menu'. You must have a solution open in order to access this dialog box.

Note: Before adding an existing project from shared directories in a multi-developer environment, you should attempt to ensure that your changes will not adversely affect or overwrite another developer's changes.

15.27.1 Look in

Use this drop-down list box to select an existing project by exploring an hierarchically structured view of files, folders, local drives, mapped network drives, and pointers to other Web resources on your computer.

> Tip: You can use the Look in drop-down list box to add, modify or remove an FTP location.

15.27.2 History

Displays the most recently modified project files and folders.

15.27.3 My Projects

Displays the files and folders at the location specified in the Projects and Solutions page in the Environment section of the Options dialog box. Projects are saved by default to n:\Documents and Settings\user profile name\My Documents\Visual Studio Projects.

15.27.4 Desktop

Displays the files and folders located on the desktop.

15.27.5 Favorites

Displays the contents of your Internet Explorer Favorites.

Tip:To add a link to your Favorites folder from this dialog box, locate the folder you want to add from the Look in drop-down list box and select Add to Favorites from the Tools drop-down menu.

15.27.6 File Name

Use this text box to filter files and folders by name. Type the full or partial file name on which to filter, using an asterisck (*) as the wildcard character. Filtering allows you to locate a file or folder whose full name or exact location is unknown.

Tip: Navigate to Web and network locations by entering the URL or network path in the File name box. For example, 'http://mywebsite' displays the files available at the 'mywebsite' Web location and '\\myserver\myshare' displays the files available at the 'myshare' location on 'myserver.' You can also open a project from a Web location by selecting Open from Web Server in the Tools menu of the Open dialog box.

15.27.7 Files of Type

Use this text box to filter project files by file extension. This drop-down list box contains all project file types for installed Visual Studio components. Filtering by file type is especially useful when searching for a project item whose exact name is not known.

Navigate Back Returns the most recently viewed folder, drive or internet location.

Up One Level Navigates the tree to the next highest folder in the hierarchy.

Search the Web Opens your Web browser to its specified search page.

Deletes the selected files or folders from storage.

New Folder Displays the New Folder dialog box. Use this option to create a new child folder under the folder selected in the Look in drop- down list box.

View By

Provides options for arranging and viewing the contents of the item selected in the Look in drop-down list box.

view
The following fields are available through the Look in drop-down list box:

List
Select to display files, folders, and other items in a list format.Details Select to display the name, size, type and last-modified date of files, folders, and other items in a list format. To sort by a particular detail, click its column header.

Properties
Select to display the names and properties of files, folders, and other items. When selected, the content pane divides in half: the left half displays the item's name and the right half displays the selected item's properties.

15.28 Preview

Select to display a preview of the file. When selected, divides the content pane in half: the left half displays the files and folders and the right half displays a preview of the file. This is useful when looking for a particular graphic file.

15.29 Arrange Icons

It has the same effect as selecting one of the column headers in Details View to sort by that column. Arrange Icons is useful when arranging items by a property other than Name (by Size, for example) in a view other than Details (Preview, for example).

Tools: Use to select a tool to apply to the item selected in the content pane

Delete: Deletes the selected file or folder from storage.
Add to Favorites: Adds the file or folder to your Internet Explorer Favorites.

Map Network Drive: Displays the Map Network Drive dialog box, which allows you to connect to a machine on a network.

Open from Web Server: Displays the Connect to Web Server dialog box, allowing you to locate and open a project that exists on a Web server.

15.30 Introduction

Now we will create a simple clock control to demonstrate using the .NET framework. The control will be a clock showing the current time, and we will leave it up to your pesonal discretion to implement a second hand and draw the clock numerals.

This assignment highlights the major points in creating a control. The reader should refer to the included source code for further details. The quickest method of creating a control from scratch is to copy one of the control samples from:

```
..\Program
Files\NGWSSDK\Samples\QuickStart\winforms\samples\Cs\WritingControls\helloworldcontrol
Copy this directory to another directory called MyControl
..\Program
Files\NGWSSDK\Samples\QuickStart\winforms\samples\Cs\WritingControls\MyControl
```

After this, rename and edit the files in this directory, change references from helloworldcontrol to myControl.

```
Helloworldcontrol.cs -> mycontrol.cs Helloworldcontrol.src ->
mycontrol.src
```

We need to add any namespaces we will be using. Namespaces contain the classes we will referencing in our control :

```
using System.ComponentModel; // Needed for control  support using
System.Timers;                    // Needed to support timer using
System.Runtime.InteropServices; // Needed for StructLayout  attribute
```

The next step is to include some extended features of C# which allows calls to the Windows operating system.

```
//    Definition    of    WINAPI    SYSTEMTIME    structure
[StructLayout(LayoutKind.Sequential)] public class SystemTime {
     public ushort wYear;
     public ushort wMonth;
     public ushort wDayOfWeek;
     public ushort wDay;
     public ushort wHour;
     public ushort wMinute;
     public ushort wSecond;
     public ushort wMilliseconds;
}
// Definition of WINAPI GetLocalTime  function[DllImport("Kernel32.dll")]
public static extern void GetLocalTime(SystemTime st);
```

15.31 Declaring the Constructor

Like Java, methods are coded inline. This takes a bit of getting used to, but makes things a lot easier to edit in the long run.

```
public MyControl() {
     m_colorHands = Color.White;
     m_colorFace = Color.Blue;
     SetStyle(ControlStyles.Opaque, false);
     SetStyle(ControlStyles.ResizeRedraw, true);
}
```

The next step is to define properties which are going to be exposed to the outside world. The new functionality included here is the attribute tag, which gives runtime information to other subsystems.

```
[ Category("Clock"), Description("Hands color for Clock"),
DefaultValue(0xFFFFFF), ] public Color HandsColor {
     get {
          return m_colorHands;
     }
     set {
          m_colorHands = value;
          Invalidate();
          Update();
     }
}
```

The code between the [ ] brackets, declares the attribute qualifiers. The get and set methods are transparent from outside of the object. To modify the color of the clock hands, you would do the following:

```
someobj.HandColor = Color.Red; The set method requires an implicit
value variable.
Overriding a base class method:
protected override void OnPaint(PaintEventArgs pe) {
     // Let base class draw its stuff first
     base.OnPaint(pe);
      // Draw code here... } Notice here the override keyword, which
simply overrides a base  class function.
```

This code calls the base class implementation of OnPaint (base.OnPaint(pe);)
Other points worth noting within the code are that objects are created on the heap. These don't require the delete operator unlike C++. The .NET framework garbage collects objects instantiated with the new operator.

Eg.

```
{
     // ... Some code
     SolidBrush brush = new SolidBrush(Color.White)
     // Scope ends... no delete operator needed for brush }
```

Another feature of C# is the changing the value of variable by the calling method.
Look at the following code:

```
* CalculatePoint(ptStart, out ptEnd,(st.wHour*5)+(st.wMinute/12),
 false, rc);
```

Note the override keyword, which simply overrides a base class function.
Notice the out method parameter. This declares that the variable has been modified when it was passed into a method.

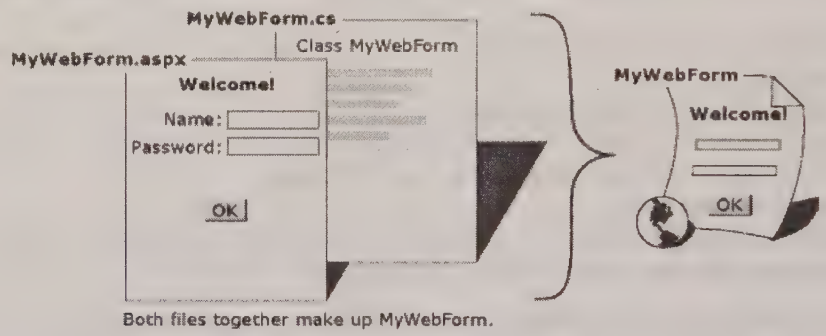
Following is the declaration:

```
protected void CalculatePoint(Point pStart, out Point pEnd,
                              int nPos, bool bFlag, Rectangle rc)
```

Mycontrol.exe hosts the control. Another way to test the control is to run WinDes.exe, then create a new C# Win32Form, then select the 'Edit/Add Library' menu item and select mycontrol.dll

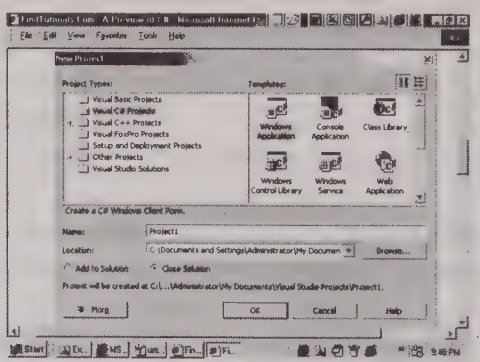
15.32 About the Samples

Before we go through the samples in this chapter, we will quickly go over how we are structuring them. Since in most cases all we want to do is perform a few operations such as writing to a file or accessing some registry keys then displaying the results, it would be simplest to provide console samples—that way we minimize the extraneous code. However, in practice, very few Windows applications are written as console applications, and it would be nice to put the output in a more realistic environment. Accordingly, the samples will all be presented as Windows applications, based on a form that has a list box, as shown here.



We haven't yet covered how to write Windows applications in C#, but we will give you just enough information here for you to understand how the samples are created.

Each project is created as a C# Windows application: So when we start up the Visual Studio developer environment and select New Project, we fill in the New Project dialog like this (though supplying our own name and path for the project).



Once the project has been created, we use the Toolbox in the developer environment to place a List box on the form—by default this will be called listBox1. The List box is actually an instance of a C# object of type ListBox, from the System.WinForms namespace.

We then examine the code the wizard has generated for us and make the following changes(shown highlighted):

```
public class Form1 : System.WinForms.Form {
/// <summary>
/// Required designer variable
/// </summary>
private System.ComponentModel.Container components;
private System.WinForms.ListBox listBox1;
private int nItems=0;
public void AddItem(string sItem)
{
   object oItem = sItem;
   listBox1.InsertItem(nItems,oItem);
   ++nItems;
}
public Form1() {
   //
   // Required for Win Form Designer support
   // InitializeComponent();
   //
   // TODO: Add any constructor code after InitializeComponent call
   //
   AddItem("An item of output");
   AddItem("2nd bit of output");
   AddItem("More output");
}
```

The code that we are adding is to the Form1 class which represents the entire form (window) created when the program is run. We first add a member field, nItems, and a member function, AddItem(). These are to make it easier for us to add lines of text to the list box. This is normally done by using the member function, InsertItem() of the ListBox class. But in its simplest overload, this function takes two parameters—the zero-based index of where we wish to insert the item, and the item itself, passed as an object (InsertItem() will use the ToString() function to extract the string to be added). Since we wish to add each successive item to the end of the list box, this would require us to keep a count of how many items we have added. Our AddItem() function automates that process for us.

```
private int nItems=0; public void AddItem(string sItem) { object oItem
= sItem; listBox1.InsertItem(nItems,oItem); ++nItems; }
```

we could also add the items to the list box with the line of code

```
listBox1.Items.Add(oItem);
```

Though either way, our function does marginally simplify the code.

We then actually add the items in the Form1 constructor. In this simplest case we have simply added three strings.

```
AddItem("An item of output");
AddItem("2nd bit of output");
AddItem("More output");
```

In all the subsequent samples these three lines will be replaced by the code to do whatever processing is required to illustrate the task in hand, followed by AddItem() calls to display the results. Each of the projects in this chapter were created in essentially the same way to the above example. Note however that in the following samples, we have also changed the name of the Form1 class and the namespace that it appears in. The developer environment by default gives us a namespace that has the same name as the project, however Microsoft guidelines say that the namespace should start with the company name (which makes sense, as you would not have any name clashes with other company's classes). So for all the projects here we will use the namespace Prog.BookSamples.CSharpPreview.ChBaseClasses.

You should bear in mind when reading through the samples that some of the classes may be modified before the .NET SDK is finally released. Hence they will give you a good idea of how to carry out tasks, but some of the names or signatures of methods may be slightly different. Also in many cases there are several classes or methods whose functionality overlaps. Just because we have presented a certain way of doing something doesn't mean there won't be alternative ways of doing the same task.

15.33 Making a clock

Now we are going to create a simple clock control to demonstrate using the .NET framework. The control will be a clock showing the current time, and it lies on your discretion to implement a second hand and draw the clock numerals.

This will highlight the major points in creating a control. You should refer to the included source code for further details. The quickest method of creating a control from scratch is to copy one of the control samples from:

```
..\Program
Files\NGWSSDK\Samples\QuickStart\winforms\samples\Cs\WritingControls\helloworldcontrol
```

Copy this directory to another directory called MyControl:

```
..\Program
Files\NGWSSDK\Samples\QuickStart\winforms\samples\Cs\WritingControls\MyControl
```

Rename and edit the files in this directory, change references from helloworldcontrol to myControl:

```
Helloworldcontrol.cs -> mycontrol.cs
Helloworldcontrol.src -> mycontrol.src
```

Edit these files and change references helloworldcontrol to myControl:

```
Hostapp.cs Makefile
```

Open a console window and type NMAKE ALL. Hopefully, two files will be produced:

MyControl.exe: The application that hosts the control MyControl.DLL—The actual control. Now the base/skeleton/boilerplate code has been created we can test it by running mycontrol.exe.

We can now start to engineer our control.

We need to add any namespaces we will be using. Namespaces contain the classes we will referencing in our control :

```
using System.ComponentModel; // Needed for control support using
System.Timers;                // Needed to support timer using
System.Runtime.InteropServices; // Needed for StructLayout attribute
```

The next step is to include some extended features of C# which allows calls to the Windows operating system.

```
// Definition of WINAPI SYSTEMTIME structure
[StructLayout(LayoutKind.Sequential)] public class SystemTime {
     public ushort wYear;
     public ushort wMonth;
     public ushort wDayOfWeek;
     public ushort wDay;
     public ushort wHour;
     public ushort wMinute;
     public ushort wSecond;
     public ushort wMilliseconds;
}
// Definition of WINAPI GetLocalTime
function[DllImport("Kernel32.dll")] public static extern void
GetLocalTime(SystemTime st);
```

Now, we declare member variables that are going to be used during the lifetime of the object:

```
private Colorm_colorHands;private  Colorm_colorFace; private
boolm_bActivateClock; private System.Timers.Timer   m_timer;
```

Notice here that the accessibility keyword is included in front of every variable declaration, unlike C++ where accessibility keywords define blocks of variables.

16

EXCEPTIONS AND ERROR HANDLING IN C#

16.1 Exception Handling in C#

The C# programming language provides a mechanism known as exceptions to help program reports and handle errors. When an error occurs, the program throws an exception. In other way, we can say that the normal flow of the program is interrupted and the runtime environment attempts to find an exception handler(a block of code that can handle a particular type of error). The exception handler can attempt to recover from the error or, if it determines that the error is unrecoverable, provides a gentle exit from the program.

Three statements play a part in handling exceptions:

1. The try statement identifies a block of statements within which an exception might be thrown.
2. The catch statement must be associated with a try statement and identifies a block of statements that can handle a particular type of exception. The statements are executed if an exception of a particular type occurs within the try block.
3. The finally statement must be associated with a try statement and identifies a block of statements that are executed whether an error within the try block occurs or not.

Here is the general form of these statements:

```
try {
    statement(s)
} catch (exceptiontype name) {
    statement(s)
} finally {
    statement(s)
}
```

Causes of exceptions

Exception can be thrown in two different ways.

1. A throw statement throws an exception immediately and unconditionally. Control never reaches the statement immediately following the throw.

2. Certain exceptional conditions that arise during the processing of C# statements and expression cause an exception in certain circumstances when the operation cannot be completed normally. An integer division operation throws a System e.g.DivideByZeroException if the denominator is zero.

16.2 The throw Statement

The throw statement throws an exception.

throw-statement:

```
throw expressionopt   ;
```

A throw statement with an expression throws the exception produced by evaluating the expression. The expression must denote a value of the class type System.Exception or of a class type that derives from System.Exception. If evaluation of the expression produces null, a NullReferenceException is thrown instead.
A throw statement with no expression can be used only in a catch block. It re-throws the exception that is currently being handled by the catch block.

Since a throw statement unconditionally transfers control elsewhere, the end point of a throw statement is never reachable. When an exception is thrown, control is transferred to the first catch clause in a try statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as exception propagation. Propagation of an exception consists of repeatedly evaluating the following steps until a catch clause that matches the exception is found. In the descriptions, the throw point is initially the location at which the exception is thrown.

In the current function member, each try statement that encloses the throw point is examined. For each statement S, starting with the innermost try statement and ending with the outermost try statement, the following steps are evaluated:

1. If the try block of S encloses the throw point and if S has one or more catch clauses, the catch clauses are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located, the exception propagation is completed by transferring control to the block of that catch clause.
2. If the try block or a catch block of S encloses the throw point and if S has a finally block, control is transferred to the finally block. If the finally block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the finally block, processing of the current exception is continued.
3. If an exception handler was not located in the current function member invocation, the function member invocation is terminated. The steps above are then repeated for the caller of the function member with a throw point corresponding to the statement from which the function member was invoked.
4. If the exception processing ends up terminating all function member invocations in the current thread or process, indicating that the thread or process has no handler for the exception, then the thread or process is itself terminated in an implementation-defined fashion.

16.3 How exceptions are handled

Exception are handled by a try statement.When an exception occurs, the system searches for the nearest catch clause than can handle the exception, as determined by the run-time type of the exception. First, the current method is searched for a lexically enclosing try statement, and the associated catch clauses of the try statement are considered in order. If that fails, the method that called try statement, and the current method is searched for a lexically enclosing try statement that encloses the point of the call to the current method. This search continues until a catch clause is found that can handle the current exception, by naming an exception class that is of the same class, or a base class, of the run-time type of the exception being thrown. A catch clause that doesn't name an exception class can handle any exception.

Once a matching catch clause is found, the system transfers control to the first statement of the catch clause. Before execution of the catch clause begins, the system first executes in order to finally clauses that were associated with try statements more nested that than the one that caught the exception.

If no matching catch clauses is found, one of the two things occurs:

1. If the search for a matching catch clause reaches a static construct or static field initializer, then a System. TypeInitializationException is thrown at the point that triggered the invocation of the static constructor. The inner exception of the TypeInitializationException contains the exception that was originally thrown.
2. If the search for matching catch clauses reaches the code that initially started the thread or program, then execution of the thread or program is terminated.

16.4 The try Statement

The try statement provides a mechanism for catching exceptions that occur during execution of a block. The try statement furthermore provides the ability to specify a block of code that is always executed when control leaves the try statement.

try-state

try block catch-clauses

try block finally-clause

try block catch-clauses finally-clause

catch-clauses:

```
specific-catch-clauses    general-catch-clauseopt specific-catch-
clausesopt    general-catch-clause specific-catch-clauses: specific-
catch-clause specific-catch-clauses   specific-catch-clause specific-
catch-clause: catch  (  class-type   identifieropt   )   block
general-catch-clause: catch   block
finally-clause: finally  block
```

There are three possible forms of try statements:

- A try block followed by one or more catch blocks.
- A try block followed by a finally block.
- A try block followed by one or more catch blocks followed by a finally block.

When a catch clause specifies a class-type, the type must be System.Exception or a type that derives from System.Exception. When a catch clause specifies both a class-type and an identifier, an exception variable of the given name and type is declared. The exception variable corresponds to a local variable with a scope that extends over the catch block. During execution of the catch block, the exception variable represents the exception currently being handled. For purposes of definite assignment checking, the exception variable is considered definitely assigned in its entire scope.

Unless a catch clause includes an exception variable name, it is impossible to access the exception object in the catch block. A catch clause that specifies neither an exception type nor an exception variable name is called a general catch clause. A try statement can only have one general catch clause, and if one is present it must be the last catch clause.

Though the throw statement is restricted to throwing exceptions of type System.Exception or a type that derives from System.Exception, other languages are not bound by this rule, and so may throw exceptions of other types. A general catch clause can be used to catch such exceptions, and a throw statement with no expression can be used to re-throw them.

An error occurs if a catch clause specifies a type that is the same as or derived from a type that was specified in an earlier catch clause. Because catch clauses are examined in order of appearance to locate a handler for an exception, without this restriction it would be impossible to write unreachable catch clauses.

Within a catch block, a throw statement with no expression can be used to re-throw the exception that was caught by the catch block. Assignments to an exception variable do not alter the exception that is re-thrown. In the example

Listing 16.1

```
class Test {
     static void F() {
          try {
               G();
          }
          catch (Exception e) {
               Console.WriteLine("Exception in F: " + e.Message);
               e = new Exception("F");
               throw;                    // re-throw
          }
     }
     static void G() {
          throw new Exception("G");
     }
     static void Main() {
          try {
               F();
          }
          catch (Exception e) {
           Console.WriteLine("Exception in Main:" + e.Message);
          }
     }
}
```

the method F catches an exception, writes some diagnostic information to the console, alters the exception variable, and re-throws the exception. The exception that is re-thrown is the original exception, so the output of the program is:

Exception in F: G Exception in Main: G

It is an error for a break, continue, or goto statement to transfer control out of a finally block. When a break, continue, or goto statement occurs in a finally block, the target of the statement must be within the same finally block, or otherwise a compile-time error occurs.

It is an error for a return statement to occur in a finally block.

A try statement is executed as follows:

- Control is transferred to the try block.

When and if control reaches the end point of the try block:

- If the try statement has a finally block,the finally block is executed.
- Control is transferred to the end point of the try statement.

The catch clauses, if any, are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general catch clause is considered a match for any exception type. If a matching catch clause is located:

- If the matching catch clause declares an exception variable, the exception object is assigned to the exception variable.
- ontrol is transferred to the matching catch block.

When and if control reaches the end point of the catch block:

- If the try statement has a finally block, the finally block is executed. Control is transferred to the end point of the try statement.
- If an exception is propagated to the try statement during execution of the catch block:
- If the try statement has a finally block, the finally block is executed.
- The exception is propagated to the next enclosing try statement.

If the try statement has no catch clauses or if no catch clause matches the exception:

- If the try statement has a finally block, the finally block is executed.

The exception is propagated to the next enclosing try statement. The statements of a finally block are always executed when control leaves a try statement.This is true whether the control transfer occurs as a result of normal execution, as a result of executing a break, continue, goto, or return statement, or as a result of propagating an exception out of the try statement. If an exception is thrown during execution of a finally block, the exception is propagated to the next enclosing try statement. If another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is further discussed in the description of the throw statement). The try block of a try statement is reachable if the try statement is reachable. A catch block of a try statement is reachable if the try statement is reachable. The finally block of a try statement is reachable if the try statement is reachable. The end point of a try statement is reachable if both of the following are true:

- The end point of the try block is reachable or the end point of at least one catch block is reachable.
- If a finally block is present, the end point of the finally block is reachable.

16.5 The System.Exception class

The System.Exception class is the base type of all exceptions. This class has a few notable properties that all exceptions share:

- Message is a read-only property that contains a human-readable description of the reason for the exception.
- InnerException is a read-only property that contains the 'inner exception' for this exception. If this is not null, this indicates that the current exception was thrown in response to another exception. The exception that caused the current exception is available in the InnerException property.
- The value of these properties can be specified in the constructor for System.Exception.

16.6 Common Exception Classes

The following exceptions are thrown by certain C# operations:

System.OutOfMemoryException: Thrown when an attempt to allocate memory (via new) fails.

System.StackOverflowException: Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.

System.NullReferenceException: Thrown when a null reference is used in a way that causes the referenced object to be required. System.TypeInitializationException Thrown when a static constructor throws an exception, and no catch clauses exists to catch in.

System.InvalidCastException: Thrown when an explicit conversion from a base type or interface to a derived types fails at run time.

System.ArrayTypeMismatchException: Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.

System.IndexOutOfRangeException: Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.

System.MulticastNotSupportedException: Thrown when an attempt to combine two non-null delegates fails, because the delegate type does not have a void return type.

System.ArithmeticException: A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.

System.DivideByZeroException: Thrown when an attempt to divide an integral value by zero occurs.

System.OverflowException: Thrown when an arithmetic operation in a checked context overflows.

16.7 The Checked and Unchecked Statements

The checked and unchecked statements are used to control the overflow checking context for integral-type arithmetic operations and conversions.

checked-statement:

```
checked   block
```

unchecked-statement:

```
unchecked   block
```

The checked statement causes all expressions in the block to be evaluated in a checked context, and the unchecked statement causes all expressions in the block to be evaluated in an unchecked context.
The checked and unchecked statements are precisely equivalent to the checked and unchecked operators, except that they operate on blocks instead of expressions.

16.8 The Lock Statement

The lock statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock. lock-statement:

```
lock   (expression)   embedded-statement
```

The expression of a lock statement must denote a value of a reference-type. An implicit boxing conversion is never performed for the expression of a lock statement, and thus it is an error for the expression to denote a value of a value-type.
A lock statement of the form lock (x) _
where x is an expression of a reference-type, is precisely equivalent to

```
System.Threading.Monitor.Enter(x);
try {
    _
}
finally {
    System.Threading.Monitor.Exit(x);
}
```

except that x is only evaluated once. The exact behavior of the Enter and Exit methods of the System.Threading.Monitor class is implementation-defined. The System.Type object of a class can conveniently be used as the mutual-exclusion lock for static methods of the class. For example:
Listing 16.2

```
class Cache {
    public static void Add(object x) {
        lock (typeof(Cache)) {
            _
        }
    }
    public static void Remove(object x) {
        lock (typeof(Cache)) {
            _
        }
    }
}
```

16.9 The Using Statement

The using statement obtains one or more resources, executes a statement, and then disposes of the resource.

using-statement:

- using (resource-acquisition) embedded-statement
- resource-acquisition:
- local-variable-declaration
- expression

A resource is a class or struct that implements System.IDisposable, which includes a single parameterless method named 'Dispose'. Code that is using a resource can call 'Dispose' to indicate that the resource is no longer needed. If 'Dispose' is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

If the form of resource-acquisition is local-variable-declaration, then the type of the local-variable-declaration must be System.IDisposable or a type that can be implicitly converted to System.IDisposable. If the form of resource-acquisition is expression then this expression must be System.IDisposable or a type that can be implicitly converted to System.IDisposable. Local variables declared in a resource-acquisition are read-only, and must include an initializer.

A using statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is enclosed in a try statement that includes a finally clause. This finally clause disposes of the resource. If a null resource is acquired, then no call to Dispose is made, and no exception is thrown. For example, a using statement of the form

```
    using (R r1 = new R ()) {
      r1.F();
}
```

is precisely equivalent to

```
R r1 = new R(); try {
      r1.F();
} finally {
      if (r1 != null) ((IDisposable)r1).Dispose();
}
```

A resource-acquisition may acquire multiple resources of a given type. This is equivalent to nested using statements. For example, a using statement of the form

```
    using (R r1 = new R(), r2 = new R()) {
      r1.F();
      r2.F();
} is precisely equivalent to: using (R r1 = new R())
      using (R r2 = new R()) {
            r1.F();
            r2.F();
      }
```

which is, by expansion, precisely equivalent to:

```
R r1 = new R();
try {
     R r2 = new R();
     try {
          r1.F();
          r2.F();
     }
     finally {
          if (r2 != null) ((IDisposable)r2).Dispose();
     }
} finally {
     if (r1 != null) ((IDisposable)r1).Dispose();
}
```

16.10 Trying and Catching

Exceptions are to be handled very carefully and in standard manner. One such method is of using try and catch keywords.The portion of code where any exception is needed is taken inside the try block. In the example the programmer expects some problem in the formula that is shown inside the try block as mathematically, it is not possible to divide any quantity by zero. The e.Message prints the type of error

Listing 16.3

```
//trying.cs
using System;
//trying.cs·
class Test {
    static int Zero = 0;
    public static void Main()
{
    // try block watches  for exceptions here
    try
    {
     int j = 22 / Zero;
    }
    // exception that occur in try block is  transferred here
    catch (Exception e)
    {
      Console.WriteLine("Exception " +e.Message);
      // e.Message prints the type of error
    }
    Console.WriteLine(" Happening after catch");
  }
}
```

The output of this program will be displayed as:

```
C:\>trying
Exception Attempted to divide by zero.
 Happening after catch
```

16.11 Hierachical Structure of Exception

The Exception is a run-time class inhibited in Common Language Runtime facility of .Net framework. 'e' is a derivation of this clause which is meant for error type exceptions. DivideByZeroException is a hierarchical class of Exception class.

Listing 16.4

```
//exception1.cs
using System; //exception1.cs
class Test {
    static int Zero = 0;
    public static void Main()
{
     try
    {
       int j = 22 / Zero;
    }
     // catch a specific exception
     catch (DivideByZeroException e)
     {
        Console.WriteLine("DivideByZero  {0}", e);
     }
     // catch any remaining exceptions
     catch (Exception e)
     {
       Console.WriteLine("Exception {0}", e);
     }
} }
```

The output of this program will be displayed as:

```
C:\>csc \exception1.cs
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.22
Copyright (C) Microsoft Corp 2000. All rights reserved.

C:\>exception1
DivideByZero  System.DivideByZeroException: Attempted to divide by zero
   at Test.Main()
```

In the following example, when the program attempts to divide by zero, an exception is generated.The runtime utility searches for the try block in AFunction() and fails to find one.So it jumps out of this function and then checks for try function in function Main(). Here one try is already available whence it looks for catch function that matches to get executed.

Listing 16.5

```
//exception2.cs
using System;
class Test {
    static int Zero = 0;
    static void AFunction()
{
    int j = 44 / Zero;
    // the following line is never executed.
    Console.WriteLine("In AFunction()"); }
    public static void Main()
{
    try
{
    AFunction();
}
    catch (DivideByZeroException e)
{
    Console.WriteLine("DivideByZero {0}", e);
} } }
```

The output of this program is displayed on the console as follows:

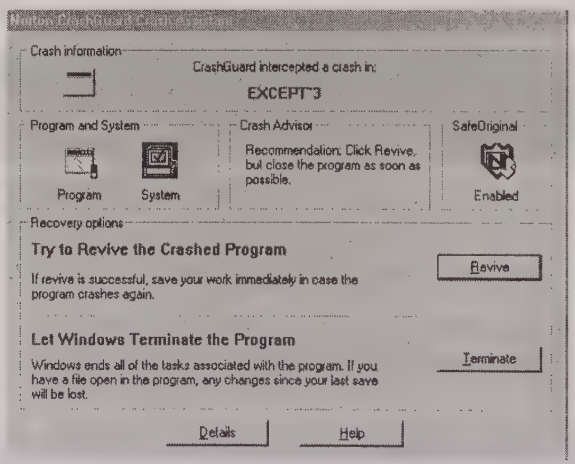
```
C:\>excep2
DivideByZero System.DivideByZeroException: Attempted to divide by zero.
   at Test.Main()
```

Listing 16.6

```
//exception3.cs
using System;
class Test {
    static int Zero = 0;
    static void AFunction()
{
    try
{
    int j = 442 / Zero;
}
    // this exception doesn't match
    catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("OutOfRangeException: {0}", e);
}
    Console.WriteLine("In AFunction()");
}
    public static void Main()
```

```
{
    try
{
    AFunction();
}
    // this exception doesn't match
    catch (ArgumentException e)
{
    Console.WriteLine("ArgumentException {0}", e);
}
}
}
```

In the program that follows, neither the catch block in AFunction nor the catch block in Main() matches the exception that is thrown. So after this, the exception is caught by the 'last chance' exception handler. The action taken by this handler depends upon how the runtime is configured, but usually it will bring a dialog box containing the information regarding the type of exception and halt the execution of the program. In computers where the Norton crashguard utility is installed, the following warning message is generated:


Crash information
CrashGuard intercepted a crash in:
EXCEPT~3
Program and System
Program
System
Crash Advisor
Recommendation: Click Revive, but close the program as soon as possible.
SafeOriginal
Enabled
Recovery options
Try to Revive the Crashed Program
If revive is successful, save your work immediately in case the program crashes again.
Revive
Let Windows Terminate the Program
Windows ends all of the tasks associated with the program. If you have a file open in the program, any changes since your last save will be lost.
Terminate
Details
Help

16.12 Passing Exceptions to Caller

In many cases, it may not be possible to do anything when an exception occurs. It is better that the calling function handles the exception.This can be done in three ways:

- Caller Beware
- Caller Confuse
- Caller Inform

16.12.1 Caller Beware

This method is a way of doing nothing to catch the exception. However, this may leave the object in undesirable state that may later on create problems.

16.12.2 Caller Confuse

The second way is not ideal way but is simpler than the third one. This can be better illustrated by following example:

Listing 16.7

```
//caller1.cs
using System;
public class Summer {
   int   sum = 0;
   int   count = 0;
   float average;
   public void DoAverage()
{
   try
{
   average = sum / count;
}
    catch (DivideByZeroException e)
{

    throw e  ;
}
} }
    class Test
{
    public static void Main()
{
    Summer summer = new Summer();
    try
{
    summer.DoAverage();
}
    catch (Exception e)
{
    Console.WriteLine("Exception: {0}", e);
} } }
```

The output of this program will be displayed as:

```
C:\>caller1
Exception: System.DivideByZeroException: Attempted to divide by zero.
   at Summer.DoAverage()
   at Test.Main()
```

This method is inclusive of catching the exception and do some cleanup and then once again throw up exception.The CSharp exceptions provide a minimal guard over the execution of the program in that after handling the exception the program should always maintain a valid state. However, this method is called Caller confuse because while the object is in a valid state after the exception occurs, the caller has often very little choice of getting information on the ways of further continuing.

16.12.3 Caller Inform

In this method information is returned to the user whenever exception occurs. Let us see this in the following program.

Listing 16.8

```
//caller.cs
using System;
public class Summer {
   int   sum = 0;
   int   count = 0;
   float average;
   public void DoAverage()
{
   try
{
   average = sum / count;
}
    catch (DivideByZeroException e)
{
    // wrap exception in another one,
    // adding additional context.
    throw (new DivideByZeroException(
      "Count is zero in DoAverage()", e));
} } }
    public class Test
{
    public static void Main()
{
    Summer summer = new Summer();
    try
{
    summer.DoAverage();
}
    catch (Exception e)
{
    Console.WriteLine("Exception: {0}", e);
} } }
```

The output of this program will be displayed as:

```
C:\>caller
Exception: System.DivideByZeroException: Count is zero in DoAverage() -
m.DivideByZeroException: Attempted to divide by zero.
   at Summer.DoAverage()
   at Summer.DoAverage()
   at Test.Main()
```

When the DivideByZeroException is caught in the DoAverage() function, it is wrapped in a new exception that gives the user additional information of expected causes of the exception.

16.13 User defined Exception Clause

The above program has the drawback that the caller cannot tell what exception occured in the call to DoAverage() by looking at the type of exception. To acknowledge that exception occured due to 'count was zero', the exception message would have to be searched for the string 'Count is zero'.This is not desirable as the user would not be able to trust that the text would remain the same in later versions of the class and the class writer would not be able to change the text. This requires a new exception clause to be created.

Listing 16.9

```
//user.cs
using System;
public class CountIsZeroException: Exception {
    public CountIsZeroException()
{ }
    public CountIsZeroException(string message)
    : base(message)
{ }
    public CountIsZeroException(string message, Exception inner)
    : base(message, inner)
{ } }
    public class Summer
{    int  sum = 0;
    int  count = 0;
    float average;
    public void DoAverage()
{
   if (count == 0)
    throw(new CountIsZeroException("Zero count in DoAverage"));
else
    average = sum / count;
} }
    class Test
{
```

```
    public static void Main()
{
    Summer summer = new Summer();
    try
{
    summer.DoAverage();
}
    catch (CountIsZeroException e)
{
    Console.WriteLine("CountIsZeroException: {0}", e);
}
}
}
```

DoAverage() in the above program determines whether there would be any exception (that is whether count is zero). and if so, creates a informatory CountZeroException and throws it.

The output of this program is displayed as:

user.cs

16.14 Error Handling Using Exceptions

The .NET Framework is the first (as far as Dr. GUI knows) multi-language framework that provides a consistent high-level error handling via structured exception handling.

We will take up structured exception handling in detail later on, but for now, note that it is a sophisticated way to allow errors, no matter where they're detected, to be handled by the code in the place where the context allows understanding and correction of the error.

For instance, let's say your program needs to allow the user to open various files. So you will want to get a file name, perhaps using the standard Windows File Open common dialog box. In addition to allowing the user to select a file from a list, this box allows the user to type a file name. Your program then takes the file name that the user types or selects and attempts to open that file.

If the file doesn't exist or if your program (or user) doesn't have appropriate permission to open the file, you have a problem.

But note that this problem won't be discovered until you are several function calls away from the call you make to open the file. For instance, you might call a helper function you write that calls another helper that finally calls File.Open. Perhaps File.Open calls some other .NET Framework internal method, and that method calls some other method. Let us just imagine that the .NET Framework calls the C Runtime function fopen(), which calls open(), which finally calls the Microsoft Windowsr API CreateFile(), which opens files for reading as well as creating new files (note that the .NET Framework probably calls CreateFile() directly.).CreateFile() probably calls some Windows internal subroutines.

Regardless of exactly how many method and function calls actually get executed before the error is discovered, the fact is that the 'file not found' error will be discovered deep within the operating system. One way or another, that error will have to be reported back to the method that originally tried to open the file. Note that only the topmost method knows what to do: Put up a dialog box telling the user that the file wasn't found, and then ask the user for another,

hopefully correctly spelled, file name. No other method or function in this whole long chain of calls knows enough about the context to actually fix the problem.

Now let's look at some of the ways of reporting and handling errors.

16.15 Ways of Reporting and Handling Errors Return Codes

Many systems and languages,including C and Windows, usually use a special return code to indicate that some sort of error has occurred. For instance, a character read function might return the character read or if you have reached end-of-file. (It's true that C and Windows support a form of structured exception handling, but that's not used in most programs and it's not a part of standard C.) COM also uses special return codes called HRESULTs.

Even systems that support structured exceptions frequently use return codes for errors and so they are compatible with other parts of the system. Until recently, exceptions were not frequently used because older implementations of exception handling made programs slower, even when an exception wasn't thrown.

Since a single number (perhaps a single value) in the return value doesn't offer much information about what caused the error, it is common to have some sort of additional information available. In Windows, you can call GetLastError to get a little bit of additional information. In Microsoft Visual Basicr, there's an Err object you can query. In COM, there is sometimes an object you can obtain by calling GetErrorInfo that implements the IErrorInfo interface to describe the error. Note that in all these cases, there is only one error object, so it represents the error state of the last operation only. Subsequent operations overwrite the previous error state.

As you can see, this medley of error handling methods doesn't lend itself to easy understanding. But more importantly, return codes are inherently unreliable.

Consider our example above. We had several layers of method calls. What if any of them 'forgot' to check for an error in a function it called and failed to pass on the error code? If that happens, the error would be lost to the calling functions and perhaps uncaught erroneous values would be passed back instead.

In unmanaged code, this problem is especially common in situations where you run out of memory. In many systems, if you attempt to allocate memory and fail, your only warning is that the pointer that's supposed to point to the memory block contains null. If you don't use this pointer right away, you won't find the error until you do. And if you are using the pointer in a context where null pointers are allowed, such as a linked list, your program might not crash or fault at all; it willl just run incorrectly. Errors should be exposed as quickly as possible, not hidden and allowed to fester.

16.16 Exceptions

Exceptions have existed in various forms for a long time. Windows has its structured exception handling (SEH) and support for it in Microsoft Visual Cr and C++. C++ uses a different, somewhat incompatible mechanism. Microsoft Foundation Classes (MFC) has two different exception handling mechanisms one for before the C++ compiler provided support for exceptions, and one that takes advantage of C++ exceptions.

16.17 What is the relevance of exceptions in this context?

Exceptions provide us with several major advantages over other types of error reporting and handling:

First, they can't be lost though a programmer forgetting to account for an error. If you don't handle the exception, it will bubble up to your caller, and their caller, and so forth. Someone is going to handle it, or your program terminates. Program termination is bad, but unhandled errors that silently corrupt data are worse. (Note, however, that exceptions can be lost if a programmer somewhere in the function call chain decides to catch and mishandle them.)

Second, they allow you to easily handle the error at the point where you have the right context to fix it. If, in any given method, you don't know how to handle a particular type of exception, but want to try,then you can and either handle it or rethrow it, perhaps adding information about your context, so that one of your calling functions can handle it.

Third, exceptions provide an exception object that contains a lot of information about the exception including an error code, text error message, help URL, and stack trace. If this information isn't enough for your error handling, you can derive a new object from an appropriate exception object and add whatever information would be helpful.

Finally, .NET Framework exceptions are cross-language and even work across machine and managed/unmanaged boundaries. Each .NET language needs to handle exceptions properly. When using a COM object, failure HRESULTs are converted to .NET Framework exceptions. When being called from a COM object, .NET Framework exceptions are converted to appropriate HRESULTs and returned. Exceptions involving Web Services and remote objects are also handled as you would expect.

16.18 Using try, catch, and throw

Most .NET languages will use some form of a try/catch statement to handle exceptions, and some form of throw statement to throw (or raise, or originate) them.

So, a simple example that deals with exceptions might look like this:

Listing 16.10

```
public static void Func1()
{
     Console.WriteLine("Starting 1st example");
   try {
      // only this code is in the protected region
      Func2();
   }
    catch (ArgumentException e) {
                  // catches this specific type
                  // and derived types
                  // uses Exception.ToString()
       Console.WriteLine("Caught: {0}", e);
   }
}
public static void Func2() {
      // this method knows nothing about
      // the exception and doesn't have to
```

```
        Func3(true);
    }
    public static void Func3(Boolean error) {
        if (error)      // check for error, throw
            throw new ArgumentException("My exception in Func3");
    }
```

The try block in Func1 contains code that might cause some sort of exception. In this case, it calls Func2, which calls Func3. Func3 throws an ArgumentException with 'My exception in Func3' as the message string. (This string should be localized into the current culture, by the way.)

The author of Func2 might have looked at the documentation for Func3 to learn that it might throw an ArgumentException although there was no clue about what to do if Func3 threw an exception. So the programmer didn't handle that exception (or any other exception) at all.

But all is not lost: The exception is passed up to the calling function, which has a chance to handle it. Note that this would have worked fine, even if the programmer of Func2 had messed up and just not paid attention to the fact that an exception might be thrown at all. Such an error would be fatal if we were relying on return values for error reporting; with exceptions, we can recover from the error without losing the exception.

In this case, Func1 does handle this specific type of exception. Its catch block prints:

```
Caught: System.ArgumentException: My exception in Func3
        at ExceptionExample1.Func1()
```

This is the type of exception, the message string, and the call stack. Note that Func2 is not listed in the call stack, even though it was clearly called. That's because the call to it was inlined. If the method were more complicated, inlining would be avoided and the method would show up in the call stack.

If you compile with debug information (csc /debug+ filename.cs), you will get more information in your stack trace, including line numbers and the "missing" method:

```
Caught: System.ArgumentException: My exception in Func3
  at ExceptionExample1.Func3() in
    C:\Foo\ExceptionSamples.cs
    :line 41 at ExceptionExample1.Func2() in
    C:\Foo\ExceptionSamples.cs
    :line 35 at ExceptionExample1.Func1() in
    C:\Foo\ExceptionSamples.cs:line 24
```

We are not showing it in this example, but if there were code in any of the methods after the point where the exception was thrown, it would have been skipped (unless it's after the try/catch block where the exception is caught and handled—for instance, at the end of Func1). The exception handling in .NET uses a termination model, where the methods that throw exceptions (or the methods that call them) are terminated, not resumed, if they don't handle the exception. (At minimum, the remainder of the try block is skipped.) The next example will show this in more detail.

There is one problem with the termination model:

16.19 What if you need to do some cleanup?

If you have cleanup code that must be executed, even if an exception is thrown, put it in the finally block, as shown in the following example.

What if there were other codes in the method? Let's analyze a more complicated example:

Listing 16.11

```
      public static void Func1() {
         Console.WriteLine("\nStarting 2nd example");
   try { // only this code is in the protected region
         Func2();
         Console.WriteLine("Never gets here");
         }
         catch (ArgumentException e) { // catches this specific type
                        // and derived types
         // uses Exception.ToString()
         Console.WriteLine("Caught: {0}", e);
         }
         catch (Exception e) { // catches any exception, use LAST!
         Console.WriteLine("General Exception: {0}", e);
         }
      finally {
         Console.WriteLine("Func1 finally: Always executed");
         }
      Console.WriteLine("Executed because exception caught");
      }

      public static void Func2() {
         try {
         Func3();
         Console.WriteLine("Never gets here");
         }
         // not same hierarchy as exception thrown, doesn't catch
         catch (IOException) {
         Console.WriteLine("Won't catch this");
         }
         finally {
         Console.WriteLine("Func 2 finally: always executed");
         }
      Console.WriteLine("Never gets here-exception not caught");
      }
      public static void Func3() {

   // this method knows nothing about
   // the exception and doesn't have to

      Func4(true);
      Console.WriteLine("Never gets here");
   // but doesn't complete
      }
      public static void Func4(Boolean error) {
         if (error)                  // check for error, throw
         throw new ArgumentException("My exception in Func4");
```

Here's the non-debug output from this program. Try to analyze and explain it.
Starting 2nd example Func 2 finally: always executed Caught: System.ArgumentException: My exception in Func4

```
   at ExceptionExample2.Func2()
   at ExceptionExample2.Func1()
```

Func1 finally: Always executed Func1: Executed because exception caught
You may try to figure it out. Go back and give it a try.

Note that this program has many lines that show 'Never gets here.' None of these print because of the termination model of exception handling. Once an exception is thrown, all the functions in between the throw and the actual catch are terminated. Then execution continues with the catch. The remaining statements in the try block are never executed if an exception is thrown.

The exception to this is if you have a finally block. The code in the finally blocks is executed as the runtime walks up the stack looking for an exception handler. Code in finally blocks is executed always—regardless of whether an exception is thrown, and regardless of whether a thrown exception is caught.

Note that the finally block in Func2 is executed before the catch block in Func1. Note also that Func2 did not catch the exception_ArgumentException is not derived from IOException, so Func2 did not catch the exception. If an IOException, or an exception derived from IOException, were thrown, Func2 would have caught it.

It is possible to have as many catch blocks as you like. We have two in Func1. They are tested in order, so put the more specific blocks first, the more general ones last. If we reversed the order here, the more specific (ArgumentException) block would never be executed. That is a common error.

Although we don't do it in this example, you could catch the exception, examine it, decide you can't fix it, and rethrow it. To rethrow the same exception object, just say throw;. If you want to throw a different exception, you can chain the exception objects together with a code like :

Listing 16.12

```
catch (FooException e) {
   if (_) { // throw chained exception
      throw new GooException("New message", e);
   else throw;    // re-throw same exception
}
```

If you throw a chained exception, the catcher can examine both exception objects: The exception it catches will be the GooException object, whose InnerException property will refer to the original FooException-derived object. This allows you to add to the exception information while preserving all of the previous information.
You will note above that you can catch an exception without declaring a variable to refer to it you will see an example of that in Func2 above. It is also possible to write a parameter-less catch block that catches all exceptions, as in:

```
catch {
   /* code here */
}
```

Note that a parameter-less catch block catches all exceptions, even those rare (and non-CLS-compliant) ones that aren't derived from Exception.

16.20 When to Use Which

While having exception handling in your code costs nothing if exceptions are never thrown, the process of figuring out which catch block gets to catch an exception as well as executing all of the finally blocks can be very expensive.

Therefore, do not use exceptions for routine flow-of-control. Use them only for exceptional or abnormal conditions. If you know the situation is going to happen, such as end of file, don't use an exception to signal it, use the return value of the method instead. You will note that in the .NET Framework, the file I/O read methods do not throw some sort of end-of-file exception. Instead, they return a value indicating that they couldn't read from the file. This is because, you're going to hit end of file just about every time you open and read a file. That's neither exceptional, nor abnormal.

But do be prepared to catch exceptions you can do something about—and you may even want to define your own exceptions.

16.21 Inside the .NET Runtime: How Exceptions Work

If you go into the innards of the .NET Runtime to explain a little bit about how exceptions work. you will get into a feature in .NET exception handling that C# doesn't support (but Visual Basic.NET does).

Exception Information Table Each executable has an exception information table that tells which blocks of code are protected (inside a try block). Each method has an array in this table; the elements of this array designate the blocks of code that are protected, any exception filters that are associated with that code, and any exception handlers (catch and finally statements).

When an exception occurs, the exception object is constructed and the runtime starts looking for a matching exception handler first in the method that's executing, then in its calling method, and so on. This is called 'unwinding the call stack'. As the stack is unwound, the local variables in each method that is terminated are discarded.

The runtime executes any finally handlers it finds while unwinding the call stack, and then executes the first matching catch block it finds (if any). After that, execution continues with the finally block, if present, and then with the statements after the try/catch(/finally) block.

Note that while unwinding the execution stack, running finally blocks, and finding the appropriate exception handler is expensive, there is no cost in memory or time if no exception is thrown.

16.22 Types of Exception Handlers

There are four types of exception handlers possible in .NET Framework methods. Note that C# supports only three of these. Visual Basic.NET supports them all.

Finally handlers: Finally handlers are executed regardless of whether an exception is thrown or a thrown exception is caught. These handlers correspond with the finally block in your code.

Fault handlers: Fault handlers are executed if an exception occurs always—in other words, they catch all types of exceptions. These correspond with a parameter-less catch block.

Type-filtered handler:Type-filtered handlers are executed if the type they specify is the same as or a base type of the exception thrown. These correspond to the catch statements used in the examples.

User-filtered handler :User-filtered handlers are executed if the expression contained in them is true. C# does not support user-filtered handlers. Visual Basic.NET supports them with the When clause of its Catch statement. In C#, you can simulate user-filtered handlers by using a type-filtered handler, checking the exception, and rethrowing the exception if you can't handle that specific exception.

16.23 The Exception Object

The exception object is, in CLS-compatible methods, derived from the class System.Exception. There are many classes derived from Exception; most do not add any members except a constructor. (An interesting exception to this rule is SecurityException, which adds members to describe the state and type of permission that threw the exception.)

We've already seen that the Exception object contains a Message (set to localized text when you throw the exception), the InnerException, which allows you to chain exceptions if necessary, and a StackTrace string, which contains the stack trace at the point the exception was thrown. It also contains a number of other interesting members:

1. HelpLink contains a string with a URI or URL for a Help file explaining the exception.
2. Source contains the name of the object or application that caused the error.

HResult contains the COM-compatible HRESULT error code.

GetBaseException returns a reference to the innermost exception in a chain. The two major interesting classes derived from Exception are SystemException and ApplicationException. (You might have noticed that all of the exception classes end in the word 'Exception.' This is by design. If you create new exception classes, they should conform to this naming convention.)

Exceptions derived from SystemException are exceptions thrown by the runtime. There are some exceptions derived from SystemException that you might want to catch in some situations, but there are many that you'll never catch—and some, such as ExecutionEngineException, which you'll probably never catch.

16.24 What Exception Should I Throw?

You should not throw most exceptions that are derived from SystemException. But there are, of course, exceptions to this rule.

Namely, you should not make up a new exception class to throw when there are perfectly good existing exception classes to use. For instance, you should throw an InvalidOperationException (or ObjectDisposedException) if your object is not in a correct state for a particular property access or method call, such as when attempting to access a file before it's opened or an object after it's had its Dispose method called.

If bad parameters are passed to a method, you should throw an ArgumentException (or subclass). If a parameter is in a bad format, you should throw a FormatException (or subclass). If a method is not implemented or supported, you should throw a NotImplementedException or NotSupportedException.

Be cautious when you add a new exception type—try to find an existing exception type that makes sense instead. If you do need to derive a new exception, you should normally derive it from ApplicationException, not Exception. Add properties only if the property has a programmatic use. Put the new exception in an appropriate namespace. If there's no need for a programmer to distinguish your exception, just use ApplicationException rather than derive a new class from it.

Using a helper function to construct your exception is a good idea, especially if you want to set more fields than just the message and inner exception. Just have your method allocate and return an exception of the appropriate type. Because the stack trace is computed when the exception is thrown, not when it's created, using a helper function won't cause the stack trace to be messed up.

Using C# with Cookies ratings: 1

A code snippet posted on aspcs about accessing and writing cookies using C#.

Listing 16.13

```
/* Read shopper id from client cookie */
public static bool GetCurrentShopperId(ref string id,
System.Web.HttpRequest request)
{
   try{
         HttpCookie cookie = request.Cookies["iparts_pa"];
         if (cookie == null){
               //cookie not found id = string.Empty;
               return false;
         }
         else{
               //cookie ok
               id = cookie.Values["shopperid"].ToString();
               //cookie empty or not
               return id.Trim().Length > 0;
         }
   }
   catch(Exception e){
         string s = e.ToString(); //to get rid of compiler warning
return false;        }
}
/* Write the shopper id to a client cookie */
public static bool SetCurrentShopperId(string shopperid,
System.Web.HttpResponse response) {
   HttpCookie cookie = new HttpCookie("iparts_pa");
   cookie.Values.Add("shopperid", shopperid);
   response.Cookies.Add(cookie);
   return true;
}
```

17

DATA HANDLING IN C#

17.1 ADO

ADO (ActiveX Data Objects) is an application-programming interface for accessing data in a connected (cursor-based) scenario. ADO provides consistent high-performance access to data and supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects using live connections to data in relational databases and other stores. ADO's primary benefits are ease of use, high speed, low memory overhead, and a small disk footprint.

ADO provides an easy-to-use interface to OLE DB, which provides the underlying access to the data.

Example

This example demonstrates how to use ADO objects from C#. It shows how you can create a dataset and add tables to it from a database. The BugTypes.mdb database used in the sample program is a Microsoft Access 2000 .MDB file.

This program accesses the BugTypes.mdb database, creates a dataset, adds the tables to it, and displays the number of tables, columns, and rows. It also displays the content of the each row.

Listing 17.1

```
000: // ADO\adosample.cs
001: using System;
002: using System.Data;
003: using System.Data.ADO;
004:
005: public class MainClass
006: {
007: public static void Main ()
008: {
009:    // set Access connection and select strings
010:     string strAccessConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=c:\\BugTypes.MDB";
011:    string strAccessSelect = "SELECT * FROM Categories";
```

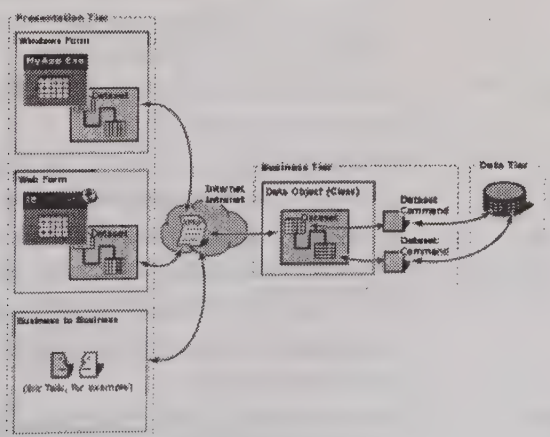
```
012:
013:    //Create the dataset and add the Categories table to it
014:    DataSet myDataSet = new DataSet();
015:    myDataSet.Tables.Add("Categories");
016:    017:    // create my Access objects
018: ADOConnection myAccessConn = new ADOConnection(strAccessConn);
019: ADODataSetCommand myAccessDataSetCmd = new ADODataSetCommand();
020: myAccessDataSetCmd.SelectCommand = new
     ADOCommand(strAccessSelect,myAccessConn);
021:
022:    myAccessConn.Open();
023:    try
024:    {
025: myAccessDataSetCmd.FillDataSet(myDataSet,"Categories");
026:    }
027:    finally
028:    {
029:    myAccessConn.Close();
030:     }
031:
032:      try
033:      {
034:       /* A dataSet can contain multiple tables,
035:       so let's get them all into an array */
036:       DataTable[] dta = myDataSet.Tables.All;
037:       foreach (DataTable dt in dta)
038:       {
039: Console.WriteLine("Found data table {0}", dt.TableName);
040:       }
041:
042:       /* The next two lines show two different ways
043:        you can get the count of tables in a dataset */
044: Console.WriteLine("{0} tables in data set", myDataSet.Tables.Count);
045: Console.WriteLine("{0} tables in data set", dta.Length);
046: /* The next several lines show how to get information
047:        on a specific table by name from the dataset */
048:       Console.WriteLine("{0} rows in Categories table",
     myDataSet.Tables["Categories"].Rows.Count);
049:       /* The column info is automatically fetched from  the
050:        database, so we can read it here */
051:  Console.WriteLine("{0} columns in Categories
     table", myDataSet.Tables["Categories"].Columns.Count);
052: DataColumn[] drc = myDataSet.Tables["Categories"].Columns.All;
053:       int i = 0;
054:       foreach (DataColumn dc in drc)
```

```
055:       {
056:        /* Print the column subscript, then the
057:         column's name and its data type */
058:    Console.WriteLine("Column name[{0}] is {1}, of
      type {2}",i++ , dc.ColumnName, dc.DataType);
059:       }
060:    DataRow[] dra = myDataSet.Tables["Categories"].Rows.All;
061:       foreach (DataRow dr in dra)
062:       {
063:        /* Print the CategoryID as a subscript,
064:         then the CategoryName */
065:     Console.WriteLine("CategoryName[{0}] is {1}", dr[0], dr[1]);
066:       }
067:      }
068:      catch (Exception e)
069:      {
070:  Console.WriteLine("Oooops. Caught an exception:\n{0}", e.Message);
071:      }
072:   }
073: }
```

The Categories table of the BugTypes.mdb database contains the following information:

Category ID	Category Name
1	Bugbash stuff
2	Appweek Bugs
3	.NET Reports
4	Internal support

When you run the sample, the following output will be displayed on the screen:



Explanation

Lines 010 and 011 set the connection to the database and the query string. Lines 014 and 015 create the dataset and add the Categories table to it. Lines 018 - 020 create the access objects. In line 022, the access connection is opened. In line 029, the connection is closed. Lines 044 and 045 show different ways to get and display the number of tables in a dataset. Line 048 - 051 show how to get and display information on a specific table from a dataset. Lines 052 - 059 read the column information and display the column subscript, name, and data type. Lines 060 - 066 read the data from the table and display the Category ID and the Category Name.

System.Data.ADO Namespace

Hierarchy diagram

The System.Data.ADO namespace is the ADO.NET managed provider.

A managed provider describes a collection of classes used to access a data source, like a database, in the managed space.

Classes	Class Description
ADOCommand	Represents a query command to be made to a data source.
ADOConnection	Represents an open connection to a data source.
ADODataReader	Provides a way of reading a for ward-only stream of data records from a data source. This class cannot be inher ited from.
ADODataSetCommand	Represents a set of data commands and a database connection which are used to fill the DataSet and update the data source.
ADOError	Collects information relevant to a warning or error re turned by the data source. This class cannot be inherited.
ADOErrors	Collects all errors generated by the adapter.
ADOException	The exception that is thrown when an error from the ADO adapter is encountered.
ADOInfoMessageEventArgs	Provides the data for the InfoMessage event. This class cannot be inherited.
ADOParameter	Represents a parameter to an ADOCommand and opionally, its mapping to DataSet columns.
ADOParameters	Collects all parameters relevant to an ADOCommand and their mappings to DataSet columns.
ADOProperties	Contains a collection of ADOProperty objects.
ADOProperty	Contains the metadata and settings for a property in an ADO data source.
ADORowUpdatedEventArgs	Provides data for the RowUpdated event. ADORowUpdatingEventArgs

	Provides data for the RowUpdating event.
ADOSchemaMappingEventArgs	Provides data for the SchemaMapping event.
Delegates	Delegate Description
ADOInfoMessageEventHandler	sents the method that will handle the InfoMessage event of an ADOConnection.
ADORowUpdatedEventHandler	sents the method that will handle the RowUpdated an ADODataSetCommand.
ADORowUpdatingEventHandler	sents the method that will handle the RowUpdating an ADODataSetCommand.
ADOSchemaMappingEventHandle	sents the method that will handle the SchemaMapping event of an ADODataSetCommand.

Enumerations:

Enumeration Description
ADODBType Specifies the data type of a field, a property, or an ADOParameter. The corresponding OLE DB type indicator is shown in parentheses in the description of each member.

17.2 Introduction to D ata Access with ADO.NET

ADO.NET is an evolutionary improvement to Microsoft ActiveX Data Objects (ADO) that provides platform interoperability and scalable data access. Because Extensible Markup Language (XML) is the format for transmitting data, any application that can read the XML format can process data. In fact, the receiving component need not be an ADO.NET component at all. It might be a Microsoft Visual Studio-based solution or any application running on any platform.

Programmability With Visual Studio.NET, you can program against your objects, not against tables and columns. ADO.NET uses strongly typed programming in which business objects Fig. prominently.

For example, consider the following line of code, using conventional (not strongly typed) programming:

```
IF TotalCost > Table("Customer").Column("AvailableCredit")
```

In this example, you are programming the ADO tables and columns in a relational, abstracted approach.

With strongly typed programming, the same example becomes much easier:

```
IF TotalCost > Customer.AvailableCredit
```

Besides being easier to read, the strongly typed code is easier to write. Specifically, automatic statement completion is sensitive to the objects you are programming. In the following example, we are able to navigate across entities in our data set, and IntelliSenser technology shows the available tables related to Customers.

```
Protected Sub Webbutton1_OnClick(ByVal sender As S
    dim mycustomer as New Customers.
End Sub
```

Orders

Fig. 17.1: Strongly Typed Programming with IntelliSense

Additionally, typed data sets run faster at execution time because the application doesn't have to search through ADO collection objects each time it needs to access a data member.

Working with Data Sets: The centerpiece of any software solution using ADO.NET is the data set. A data set is an in-memory copy of database data. A data set contains any number of data tables, each of which typically corresponds to a database table or view. A data set constitutes a "disconnected" view of the database data. That is, it exists in memory without an active connection to a database containing the corresponding tables or views. This disconnected architecture enables greater scalability by only using database server resources when reading or writing from the database.

At run time, data will be passed from the database to a middle-tier business object and then down to the user interface. To accommodate the exchange of data, ADO.NET uses an XML-based persistence and transmission format. That is, to transmit data from one tier to another, an ADO.NET solution expresses the in-memory data (the data set) as XML and then sends the XML to the other component.

The following illustration shows the major components of an ADO.NET solution.

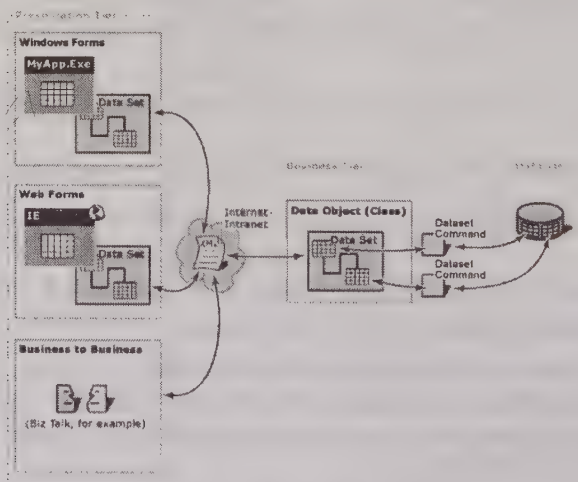


Fig. 17.2: ADO.NET Architecture

To make working with data easy in Visual Studio.NET, there are many new features. For the hardcore XML developer, Microsoft has included a rich color-coded XML Designer with statement completion and tag completion.

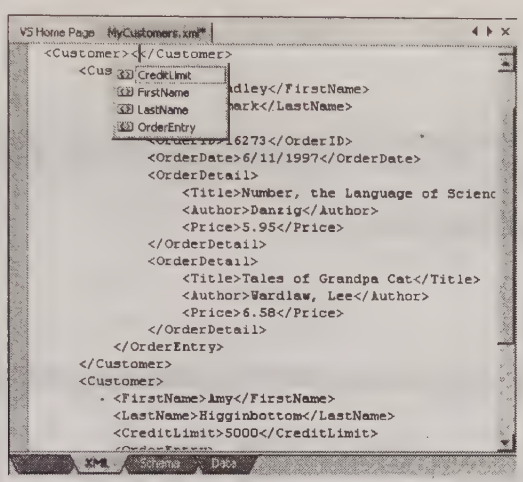


Fig. 17.3: Color-Coded XML Designer with Statement and Tag Completion

For a more graphic view of data, developers can use the design view of the XML Designer. Simply drag and drop tables from any data source, including Microsoft SQL Server™ and Oracle databases, from the Server Explorer to the design surface. You can create data sets that are made up of data from numerous sources including any XML file.

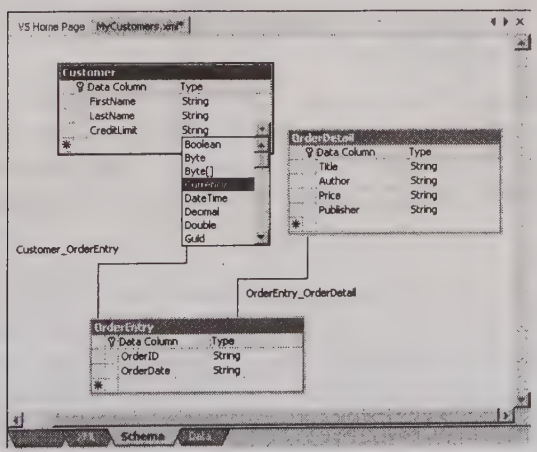


Fig. 17.4: ADO.NET Data Set Designer

Often you need to add, modify, or delete data while you are designing your application. With the Data Preview tab, you can not only add and modify data, but also navigate the relationships of your data.

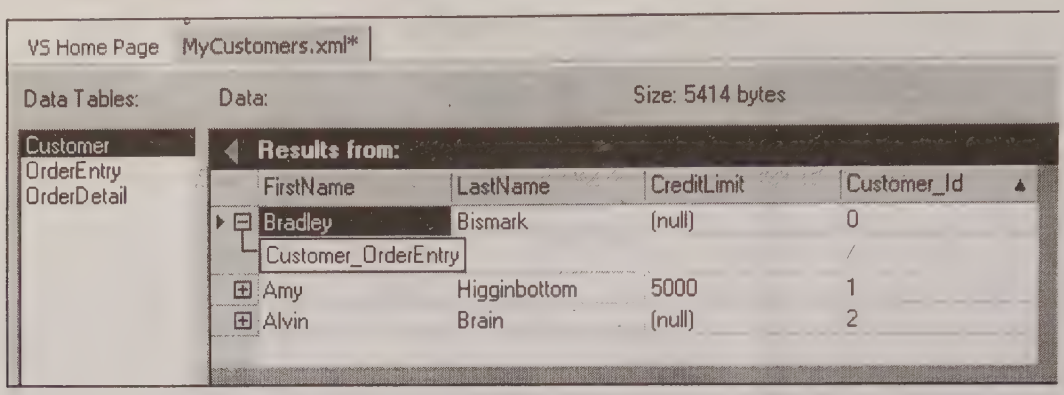


Fig. 17.5: Data Preview

The data-binding technologies for Visual Studio.NET have been dramatically improved to take full advantage of ADO.NET. Consequently, building user interfaces that interact with data is easy. More importantly, now you can bind values to business objects and Web Services.

Data binding in ASP+ allows us to declaratively and easily bind data from a data source to an ASP+ Server Control in a way that is similar to Internet Explorer's client-side data binding (but remains cross-browser compatible).

Data binding in ASP+ is pretty easy. You simply bind a data source to a server-side Web control, specify a few properties and ASP+ takes care of displaying the data. It is a great way to display grids, lists, labels, text boxes, options, checkboxes, etc., and the best part is that you don't have to do any looping or building of html tags.

ASP+ has a DataGrid class which helps to bind to large amounts of data easily. This article will describe how to connect to a SQL Server database and bind data using the DataGrid object. We will perform a simple SELECT* of the authors table in the pubs database to get all the rows and columns. Later on in this book, we will see how to do something with the data once we have it.

Three Easy Steps

For this example,we will be performing all data access in a server side script block located directly in the aspx page. Alternatively, we could use a code behind file or more likely a data access component residing on another tier in our architecture. Keep an eye out for future articles discussing both of these options. For now, here are the basic steps you need to bind data to the DataGrid:

1. Import System.Data and System.Data.SQL namespaces

<%@ Import Namespace="System.Data" %>

<%@ Import Namespace="System.Data.SQL" %>

csharpindex.com/colorCode

2. Override the Page_Load event in a server side script block using the .Net language of your choice (I prefer C#). I'm just doing a basic FillDataSet with a Command object to get the data (more information about data access with ADO.NET see 'Powerful Data Access the .NET Way', 'ADO.NET DataSets, Recordsets on Steroids', and 'The Power Of Managed Providers').

Listing 17.2

```
<SCRIPT LANGUAGE="C#" RUNAT="SERVER"> DataView dv;
  // Fires every time the page is loaded.
  protected void Page_Load(Object src, EventArgs e){
    // Create and open a connection to our data source.
    SQLConnection con = new
SQLConnection("server=localhost;uid=sa; pwd=;database=pubs");
    con.Open();
 // Create a new Command object, specifying
 // the data command we want executed and a valid connection.
  SQLDataSetCommand cmd = new SQLDataSetCommand("SELECT *
FROM Authors ORDER BY au_lname", con);
    // Create a new DataSet object for the Command
    // object to fill for us.
    DataSet ds = new DataSet();
    cmd.FillDataSet(ds, "AuthorList");
    // Create a new DataView object and coerce it
    // to a Collection type so our DataGrid can use it.
    ICollection dv = new DataView(ds.Tables[0]);
    // Specify where the DataGrid is to get it's data.
    DataGrid1.DataSource = dv;
 // Explicitly call DataBind which will cause the OnDataBind event
 // to be raised on each server control and handled accordingly.
    DataGrid1.DataBind();
  }
</SCRIPT>
csharpindex.com/colorCode
```

3. In the body of the aspx page, use an ASP:DataGrid tag.

```
<ASP:DataGrid id="DataGrid1" runat="server" />
  csharpindex.com/colorCode
```

Specify A Custom Look

This is pretty plain. The DataGrid class has a whole slew of properties we could have implemented. For the sake of simplicity, we kept this first example simple. By specifying a few of the DataGrid's properties, we can dramatically change the look and feel of our grid withought writing any real code.

```
<ASP:DataGrid id="DataGrid1" runat="server"
  WIDTH="650" backcolor="#ffffcc"
  bordercolor="black"
```

```
    showfooter="false"
    cellpadding="5"
    cellspacing="0"
    Font-Name="Tahoma"
    Font-Size="8pt"
    HeaderStyle-BackColor="#cccccc"
    maintainstate="false"/>
csharpindex.com/colorCode
```

Obviously, what's going on is that ASP+ is creating an HTML table for us, behind the scenes. By providing values for a few properties, we are telling ASP+ to use certain styles, colors, etc. Let me break down exactly what we specified. On the first line, we simply give the grid an ID so we can reference it from our code. We also include the mandatory runat=server property. This tells the ASP+ parser that something needs to be done on the server and not to simply display it as HTML. On the next line, we specify the table width and background color, followed by a border color. The ShowFooter property is a little odd. If you pass it a value of 'true' you simply get an empty row of cells at the bottom of the table. If you have done any HTML before (and I know you have) the next few lines are pretty self explainatory. The last property, maintainstate requires a bit more explaination. If we bind to an editable Web control, we can specify that we want that control to 'save' any values we enter into that control so that if we leave the page and hit the back button, all of our values will remain.

The ASP+ DataGrid object provides a fast, easy way to access large amounts of data. Using a SQL Server database, we have seen as to how to grab all rows and columns in the Authors table of the pubs database and quickly display them. Additionally, we looked at some of the properties of the DataGrid that can be used to customize the look and feel of the user interface. This is only a small fraction of what is possible with ASP+ DataBinding.

As you develop applications using ADO.NET, you will have different requirements for working with data. In some cases, you might simply want to display data on a form. In other cases, you might need to devise a way to share information with another company.

No matter what you do with data, there are certain fundamental concepts that you should understand about the data approach in ADO.NET. You might never need to know the some of the details of data handling—for example, you might never need to directly edit an XML file containing data—but it is very useful to understand the data architecture in ADO.NET, what the major data components are, and how the pieces fit together.

This topic presents a very high-level overview of these most important concepts. The topic deliberately skips over many details—for example, there is much more to datasets than what is mentioned here—in favor of simply introducing you to ideas behind data integration in ADO.NET.

17.3 ADO.NET Uses a Disconnected Data Architecture

In traditional client-server applications, components establish a connection to a database and keep it open while the application is running. For a variety of reasons, this approach is impractical in many applications:

1. Open database connections take up valuable system resources. In most cases, databases can maintain only a small number of concurrent connections. The overhead of maintaining these connections detracts from overall application performance.
2. Similarly, applications that require an open database connection are extremely difficult to scale up. An application that does not scale up well might perform acceptably with four users will likely not do so with a hundred. Web applications in particular need to be easily scalable, since the traffic to a Web site can go up by orders of magnitude in a very short period.

In Web applications, the components are inherently disconnected from each other. The browser requests a page from the server; when the server has finished processing and sending the page — it has no further connection with the browser until the next request. Under these circumstances, maintaining open connections to a database is not viable, since there is no way to know whether the data consumer (the client) requires further data access. A model based on connected data can make it difficult and impractical to exchange data across application and organizational boundaries using a connected architecture. If two components need to share the same data, both have to be connected, or a way must be devised for the components to pass data back and forth. For all these reasons, data access with ADO.NET is designed around a disconnected architecture. Applications are connected to the database only long enough to fetch or update the data. Because the database is not hanging on to connections that are largely idle, it can service many, many more users.

There are times when it is still useful to work with connected data. For example, in an application that performs a high volume of updates with possible record contention, a connected data architecture can be very important. A typical scenario is a ticket-booking application, where the users need to work with information that is up to the moment. For these types of applications, you would probably design your data access around ADO.

17.4 Data is Cached in Datasets

Far and away the most common data task is to retrieve data from the database and do something with it: display it, process it, or send it to another component. Very frequently, the application needs to process not just one record, but a set of them: a list of customers or today's orders, for example. Often the set of records that the application requires comes from more than one table: my customers and all their orders; all authors named 'Smith' and the books they've written; and other, similar, sets of related records.

Once these records are fetched, the application typically works with them as a group. For example, the application might allow the user to browse through all the authors named 'Smith' and examine the books for one Smith, then move to the next Smith, and so on.

In a disconnected data model, it is impractical to go back to the database each time the application needs to process the next record. (Doing so would undo much of the advantage of working with disconnected data in the first place.) The solution, therefore, is to temporarily store the records retrieved from the database and work with this temporary set.

This is what a dataset is. A dataset is a cache of records retrieved from the database. It works like a virtual data store— it includes one or more tables based on the tables in the actual database, and it can include information about the relationships between those tables and constraints on what data the tables can contain.

The data in the dataset is usually a much-reduced version of what is present in the database. However, you can work with it in much the same way you do the real data. While you are doing so, you remain disconnected from the database, which frees it to perform other tasks.

Of course, you often need to update data in the database (although not nearly as often as you retrieve data from it). You can perform update operations on the dataset, and these are written through to the underlying database.

An important point is that the dataset is a passive container for the data. To actually fetch data from the database and (optionally) write it back, you use dataset commands. A dataset command contains the instructions for how to populate a single table in the dataset and how to update the corresponding table in the database. The instructions are methods that encapsulate either SQL commands or references to stored procedures. Thus, a dataset command's FillDataSet method might execute a SQL statement such as SELECT au_id, au_lname, au_fname FROM authors whenever the method is called.

Because a dataset is effectively a private copy of the database data, it does not necessarily reflect the current state of the database. If you want to see the latest changes made by other users, you can refresh the dataset by calling the appropriate FillDataSet method.

One of the advantages of using datasets is that components can exchange them as required. For example, a business object in the middle tier might create and populate a dataset, then send it to another component elsewhere in the application for processing. This facility means that components don't have to individually query the database.

17.5 Datasets are Independent of Data Sources

Although a dataset acts as a cache for data drawn from a database, the dataset has no actual relationship with the database. The dataset is a container; it is filled by methods such as the SQL commands or stored procedures executed from a dataset command.

Because a dataset is not directly tied to a data source, it is a good integration point for data coming from multiple sources. For example, some of the data in a dataset might come from your database, whereas other parts of it might come from a different database or a non-database source such as a spreadsheet. Some of the data in a dataset might arrive in a stream sent by another component.

Once the data is in a dataset, you can work with it using a consistent object model, regardless of its original source.

17.6 Data is Persisted as XML

Data needs to be moved from the data store to the dataset, and from there, to various components. In ADO.NET, the format for transferring data is XML.

Similarly, if data needs to be persisted (for example, into a file), it is stored as XML. If you have available an XML file, you can use it like any data source and create a dataset out of it.

In fact, in ADO.NET, XML is a fundamental format for data. The ADO.NET data APIs automatically create XML files or streams out of information in the dataset and send them to another component. The second component can invoke similar APIs to read the XML back into a dataset.

17.6.1 Advantages of using XML

Basing data protocols around XML offers a number of advantages:

1. XML is an industry-standard format. This means that your application data components can exchange data with any other component in any other application, as long as that component understands XML. Many applications are being written to understand XML, which provides an unprecedented level of exchange between disparate applications.
2. XML is text-based. The XML representation of data uses no binary information, which allows it to be sent via any protocol, such as HTTP. Most firewalls block binary information, but by formatting information in XML, components can still easily exchange the information.
3. For most scenarios, you do not need to know XML in order to use data in ADO.NET. If your goal is simply to read and write data to your database, the XML used by ADO.NET is invisible to you—as invisible as the format by which your browser and Web server exchange information. ADO.NET automatically converts data into and out of XML as needed; you interact with the data using ordinary programming methods.

17.7 Schemas Define Data Structures

Although you don't need to know anything about XML to read and write to the database and work with datasets, there are situations in which working with XML is precisely the goal you are after. These are situations in which you are not accessing data, but instead, working with the design of data. To put it another way, in ADO.NET, you use XML directly when you are working with metadata.

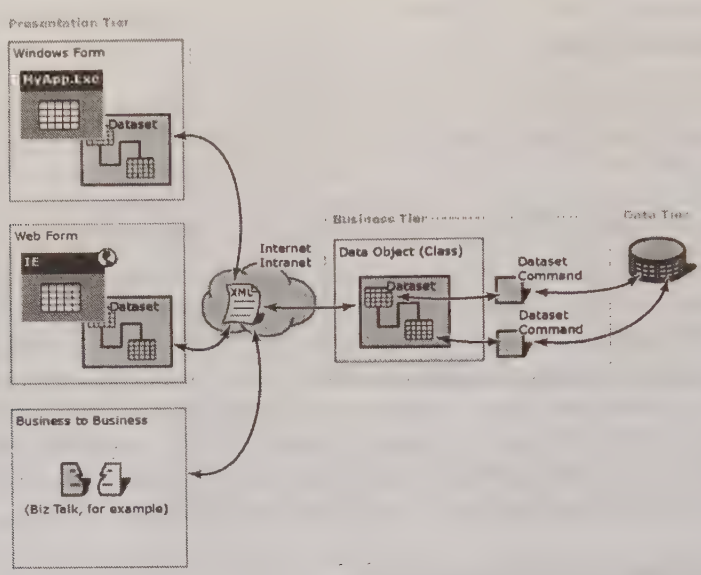
Datasets are represented as XML. The structure of the dataset—the definition of what tables, columns, data types, constraints, and so on are in the dataset—is defined using an XML schema.

For most of the work you do with data in ADO.NET, you do not have to delve deeply into schemas. Typically, the Visual Studio.NET tools will generate and update schemas as needed based on work you do in visual designers. For example, when you use the tools to create a dataset representing tables in your database, Visual Studio.NET generates an XSD schema describing the structure of the dataset.

However, there are times when you want to create or edit schemas yourself. A typical example is developing a schema in conjunction with a partner or client. In that case, the schema serves as a contract between you and the partner regarding the shape of the XML-based data you will exchange. In that case, you will often have to map the schema elements to the structure of your own database.

17.8 Components of ADO.NET

The following illustration shows the major components of an ADO.NET application.

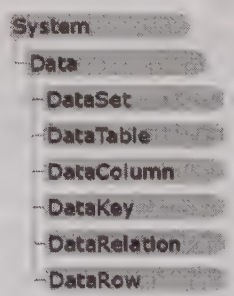


17.9 Programming Architecture

Datasets store data in a disconnected cache. The structure of a dataset is something like that of a database; it exposes a hierarchical object model of tables, rows, and columns. In addition, it contains constraints and relations defined for the dataset.

You can create and manipulate datasets using the following portions of the .NET Framework namespaces.

17.10 DataSet Namespace



The fundamental parts of a dataset are exposed to you through standard programming constructs like properties and collections.

For example:

1. The DataSet class includes the Tables collection of data tables and the Relations collection of relations
2. The DataTable class includes the Rows collection of table rows, the Columns collection of data columns, and the ChildRelations and ParentRelations collections of data relations.
3. The DataRow class includes the RowState property, whose values indicate whether and how the row has been changed since the data table was first loaded from the database. Possible values include Deleted, Modified, New, and Unchanged.

17.11 Typed versus Untyped Datasets

Datasets can be typed or untyped. A typed dataset is a dataset subclassed from the base DataSet class, and that has a schema file (an .xsd file) that describes the structures of the tables that the dataset contains. The schema contains the table and column names, the data types of the information in the columns, and information about constraints on the data. An untyped dataset, in contrast, has no corresponding schema.

You can use either type of dataset in your applications. However, typed datasets make programming with the dataset easier and less error-prone. The typed dataset generates an object model in which its tables and columns become first-class objects in the object model. For example, if you are working with a typed dataset, you can reference a column using syntax such as the following:

```
//C#
// This accesses the title_id column in the first
// row of the  titles table

s = dsPubs1.titles[0].title_id;
```

In contrast, if you are working with an untyped dataset, the equivalent syntax is:

```
//C#

s = dsPubs1.Tables["titles"][0].Columns["title_id"];
```

In addition to being easier to work with, the syntax for the typed dataset provides type checking. This greatly reduces the possibility of errors in assigning values to dataset members.

17.12 Related Tables and Relation Objects

If you have multiple tables in a dataset, the information in the tables might be related. However, unlike a database, a dataset has no knowledge of such relationships. To work with data in related tables, therefore, you can create relations (DataRelations) that describe the relations between the tables in the dataset. Relations can be used to programmatically navigate from a parent row to its related child rows, and from a child row back to its parent.

For example, imagine data about school courses. The data might contain a table containing instructor information such as the following:

InstructorID	Name	HireDate
i444	Frank Miller	09/01/1992
i777	Linda Kobara	09/01/1998
i123	Salman Mughal	09/01/1983

The dataset might also contain another table with class information. The classes table contains an instructor ID as a foreign key column, which might then look like the following:

Department	Course Number	Name	InstructorID
Science	999	Biology	i444
Science	101	Physics	i777
Math	101	Calculus	i777

Because instructors teach more than one class, there is a one-to-many relationship between instructors and classes. For example, in the table above, the instructor i777 (Linda Kobara) teaches two classes.

You use data relations to move from a particular row of one data table to the corresponding rows of another data table. You can navigate a data relation in either direction. For example, you can move from the row describing Frank Miller to the collection of rows describing the courses he teaches, or you can move from the row describing the course Science 999 (Biology) to the row describing its teacher (Frank Miller).

17.13 Updating Datasets and Data Stores

When changes are made to records in the dataset, the changes have to be written back to the database. The DataRow class used to manipulate individual records includes the RowState property, whose values indicate whether and how the row has been changed since the data table was first loaded from the database. The possible values include Deleted, Modified, New, and Unchanged.

To write changes from the dataset to the database, you call the Update method of the appropriate dataset command. That method then examines the value of the RowState property to determine which records need to be written to the database and what specific database command (add, edit, delete) should be invoked.

Dataset commands read data into datasets and transmit changes back to the database. The dataset command contains a SQL statement or reference to a stored procedure that can be invoked to read or write to the database.

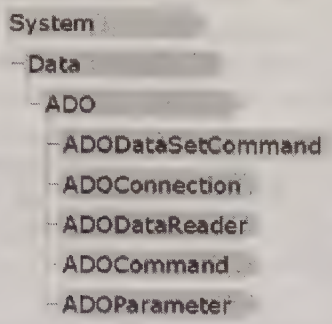
There are two types of dataset command objects.They are:

1. The ADODataSetCommand object which is suitable for use with any data source exposed by an OLE DB provider.

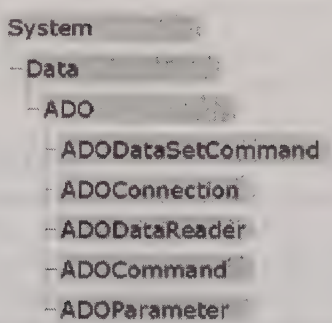
2. The SQLDataSetCommand object which is specific to the SQL Server. Because it does not have to go through an OLE DB layer, it is faster than the ADODataSetCommand. However, it can only be used with the SQL Server 7.0 or later.

You can create and manipulate dataset commands using the following portions of the .NET Framework namespaces.

ADODataSetCommand Namespace



SQLDataSetCommand Namespace



Each DataTable is associated with a different dataset command. When you want to populate the dataset, you call one of the dataset command's methods, which executes the SQL statement or stored procedure. The results of the statement or procedure are read by a DataReader, which in turn fills them into the associated table in the dataset.

Similarly, when you want to update the database, you invoke a method of the dataset command. The method calls the appropriate SQL statement or stored procedure to make the actual update in the database.

17.14 Connection Objects

A dataset command can have an open connection to a data source. An ADOConnection is a connection to any OLE DB data source. An SQLConnection is a connection to a SQL Server data source. In both cases, the connection object represents a unique session within the data source.

Both ADOConnection and SQLConnection objects provide methods to establish and modify connection properties such as user ID and password, isolation level, and connection timeout setting. They also give you methods to begin, commit, and rollback database transactions.

17.15 Table Mappings

By default, when you generate a dataset from database tables, the names of the tables and columns are the same in the dataset as in the database. However, you might find this impractical. For example, you might find the names used in the database too terse or verbose; or the names might be in a foreign language. If you are working with an existing schema, you might also find that the names defined in the schema do not match those used in your database.

Therefore, the names in the database and dataset do not have to match. Instead, you can create new names in the dataset command, and then map those to the names used in the database. The ADODatSetCommand and SQLDataSetCommand use the TableMappings collection to maintain the correspondence between structures of the dataset (data tables and data columns) and structures of the data store (tables and columns).

17.16 Reading and Updating with DataSet Commands

You can use a dataset command to undertake the following operations:

1. Retrieve rows from a data store into corresponding data tables within the dataset.
2. To retrieve rows into a dataset, use the FillDataSet method on the SQLDataSetCommand object or the ADODataSetCommand object. When you invoke the method, it transmits an SQL SELECT statement to the data store.
3. Transmit the contents of a datatable to the corresponding data store.
4. To transmit a datatable of the dataset to the data store, use the Update method on the SQLDataSetCommand object or the ADODataSetCommand object. When you invoke the method, it transmits whatever SQL INSERT, UPDATE or DELETE statements are needed. The method inspects the RowState property of each DataRow. If the row is new, the method transmits an SQL INSERT statement. If the row is modified, it transmits an SQL UPDATE statement. If the row is deleted, it transmits an SQL DELETE statement.

Every ADODataSetCommand and SQLDataSetCommand object has four properties containing information about how to read and update a data source:

1. **SelectCommand** — Gets or sets a command for retrieving rows from the data store.
2. **InsertCommand** — Gets or sets a command for inserting rows into the data store.
3. **UpdateCommand** — Gets or sets a command for modifying rows in the data store.
4. **DeleteCommand** — Gets or sets a command for deleting rows from the data store.

The properties are themselves objects —instances of the ADOCommand or SQLCommand class. These classes include the CommandText property containing the SQL statement to be transmitted to the database.

Although you can explicitly set the text of an ADOCommand or SQLCommand, you do not always need to. In many situations, the ADODataSetCommand or SQLDataSetCommand can automatically generate the SQL statements it needs.

As an illustration of how updates are made, suppose your program uses a dataset containing a single data table. The program retrieves two rows from the database. After the retrieval, the in-memory data table looks like the following:

(RowState)	CustomerID	Name	Status
(Unchanged)	c200	Pat Coleman	Good
(Unchanged)	c400	Ketan Dalal	Pending

Your program changes Ketan Dalal's status to "Preferred." As a result, the value of the DataRow.RowState property for that row changes from Unchanged to Modified. The value of the RowState property for the first row remains Unchanged. The data table now looks like this:

(RowState)	CustomerID	Name	Status
(Unchanged)	c200	Pat Coleman	Good
(Modified)	c400	Ketan Dalal	Preferred

You program now calls the Update method to transmit the in-memory modifications to the database. The method inspects each row in turn. For the first row, the method transmits no SQL statement to the database, because that row is unchanged since it was retrieved from the database.

For the second row, however, the Update method automatically constructs the appropriate SQL statement and transmits it to the database. The specific syntax of the SQL statement depends on the dialect of SQL supported by the underlying data store. But the following general traits of the transmitted SQL statement are noteworthy:

1. The transmitted SQL statement is an UPDATE statement. The ADODataSetCommand or SQLDataSetCommand knows to use an UPDATE statement because the value of the RowState property is Modified.
2. The transmitted SQL statement includes a WHERE clause indicating that the target of the UPDATE statement is the row whose CustomerID = 'c400'. This part of the SELECT statement distinguishes the target row from all others because the CustomerID is the primary key of the target table.
3. The transmitted SQL statement includes the SET clause, to set the new values of the modified columns.

The resulting SQL might look like this:

```
UPDATE Customer_Table
  SET Cust_Status = 'Preferred'
  WHERE Cust_ID = 'c400'
```

17.17 Concurrency Control

Since datasets are disconnected, you do not hold locks on records in the database. Therefore, if you want to update the database, and if it is important for your application to maintain concurrency control, you must reconcile records in the dataset with those in the database. For example, you might find that records in the database have changed. In that case, you must execute application-appropriate logic to specify what should happen with the database record or the changed record you have in your dataset.

The nature of ADO.NET connections makes it impractical to support pessimistic concurrency. However, you can implement optimistic concurrency with dataset commands.

17.18 ADOCommand and SQLCommand

When the ADODataSetCommand or SQLDataSetCommand transmits an SQL statement to the data store, you can control the details of that transmitted statement. For example, when you call the FillDataSet method, you can control exactly what the transmitted SQL SELECT statement looks like.

You have this control because every ADODataSetCommand or SQLDataSetCommand includes the SelectCommand property, which you use to get or set a command for retrieving rows from the data store. When the dataset command transmits an SQL SELECT statement to the data store, it uses the value of the SelectCommand property.

17.19 ADODataReader and SQLDataReader

If your program needs to perform a sequential, read-only pass through a query result, you can use a dataset command to establish a reader object. That is, you can establish an ADODataReader or an SQLDataReader.

An ADODataReader or an SQLDataReader includes properties and methods that help you scan through the query result. For example, the NextResult method advances to the next row of the query result; the Close method destroys the DataReader object, and the HasMoreRows property indicates whether there are more rows to retrieve.

To establish an ADODataReader or SQLDataReader object, you must first insert the SQL SELECT statement into the CommandText property of an ADOCommand or SQLCommand, the invoke the Execute method on that ADOCommand or SQLCommand.

17.20 Databases

Database is an organised collection of related data. Following are the functions that are normally reqiured in database maintenance :

- Create database
- Store data
- Retrieve data
- Sort data
- Update data
- Relate with other databases

17.21 Objectives behind maintaining databases:

- integrate existing data files
- share data among users
- simplify changes
- accuracy, consistency
- low cost
- security
- central control

17.22 Relational database

A relational database is a collection of data items organized as a set of formally-described tables from which data can be accessed or reassembled in many different ways without having to reorganize the database tables. The relational database was invented by E. F. Codd at IBM in 1970.

The standard user and application progam interface to a relational database is the structured query language (SQL). SQL statements are used both for interactive queries for information from a relational database and for gathering data for reports.

In addition to being relatively easy to create and access, a relational database has the important advantage of being easy to extend. After the original database creation, a new data category can be added without requiring that all existing applications be modified.

A relational database is a set of tables containing data fitted into predefined categories. Each table (which is sometimes called a relation) contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns. For example, a typical business order entry database would include a table that described a customer with columns for name, address, phone number, and so forth. Another table would describe an order: product, customer, date, sales price, and so forth. A user of the database could obtain a view of the database that fitted the user's needs. For example, a branch office manager might like a view or report on all customers that had bought products after a certain date. A financial services manager in the same company could, from the same tables, obtain a report on accounts that needed to be paid.

When creating a relational database, you can define the domain of possible values in a data column and further constraints that may apply to that data value. For example, a domain of possible customers could allow up to ten possible customer names but be constrained in one table to allowing only three of these customer names to be specifiable.

The definition of a relational database results in a table of metadata or formal descriptions of the tables, columns, domains, and constraints.

17.23 SQL (Structured Query Language)

SQL (Structured Query Language) is a standard interactive and programming language for getting information from and updating a database. Although SQL is both an ANSI and an ISO standard, many database products support SQL with proprietary extensions to the standard language. Queries take the form of a command language that lets you select, insert, update, find out the location of data, and so forth. There is also a programming interface.

SQL is the standard language for accessing relational databases such as Oracle, DB2, Informix, Sybase, and Teradata. With the growth of the World Wide Web and the information explosion that comes with it, knowldge of SQL proves to be invaluable for anyone in the IT field because this vast amount of information necessarily needs to be stored somewhere, and most of the time some flavor of relational database is the choice.

17.24 Microsoft's SQL Server

SQL Server 2000 includes many new features that extend its capabilities as a high performance relational database system with a rich development environment. Microsoft SQL Server 2000 provides major enhance-ments to the current SQL Server product. Numerous changes have been made across the product, including the Relational Engine, the Storage Engine, administration and tools, replication, the Analysis Service (OLAP Services and Data Mining), English Query, full-text search, integration with Windowsr 2000 (with Active Directory™ , Address Windowing Extensions, and Windows 2000 Data Center), and Meta Data Services (including the Repository Engine).

The main areas of interest for developers will be the improvements in the Relational Engine. These features extend the functionality of the database platform to make SQL Server 2000 an outstanding, feature-rich environment for application development.

17.25 Accessing Data with ADO.NET

ADO.NET is the successor to Microsoftr ActiveXr Data Objects (ADO). It is a W3C standard-based programming model for creating distributed, data-sharing applications. Like ADO, ADO.NET is an application programming interface (API) to access data and information. ADO.NET supports a variety of development needs, including the creation of front-end database clients and middle-tier business objects that use applications, tools, languages, or Internet browsers.

17.26 Building .NET Applications

Accessing Data

Accessing data has become a major programming task for modern software programming, both for standalone applications and for web-based applications. Microsoft's ADO.NET technology offers a solution to many of the problems associated with data access.

ADO.NET is an evolutionary improvement to Microsoft ActiveX Data Objects (ADO). It is a standards-based programming model for creating distributed, data-sharing applications. ADO.NET offers several advantages over previous versions of ADO and over other data access components. These benefits fall into the following categories: Interoperability, Maintainability, Programmability, and Performance. The topics in this section will further describe these benefits and explain how to use ADO.NET.

17.27 Connecting to Database

For performing any operation on database we need to connect to database. This is done automatically through ADO connection, of which code is associated with VS.net Programming environment.

The program in listing 17.3 is an example giving this connectivity.

Listing 17.3

```
  Connecting Database
//sharp1.cs
class xyz1 {
   public static void Main()
{
   try
{
   System.Data.ADO.ADOConnection s;
   s=new System.Data.ADO.ADOConnection();
   System.Console.WriteLine("C-Sharp");
}
   catch(System.Exception e)
{
   System.Console.WriteLine(e.ToString());
} } }
```

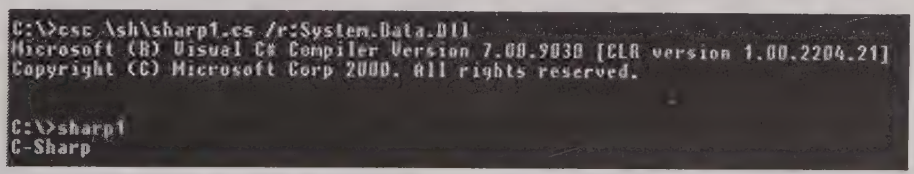
Above program can be compiled through command line at DOS prompt giving command line option as /r:System.Data.Dll

Because the code for ADO.Connection is available at this Dll.

I have saves the program sharp1.cs in directory c:\sh, so for compilation i have given the command as

```
csc \sh\sharp1.cs  /r:System.Data.Dll
```

The output of this program shall be displayed on console as:


C:\>csc \sh\sharp1.cs /r:System.Data.Dll
Microsoft (R) Visual C# Compiler Version 7.00.9030 [CLR version 1.00.2204.21]
Copyright (C) Microsoft Corp 2000. All rights reserved.
C:\>sharp1
C-Sharp

The above program consists of an object s resembling as System.Data.ADO.ADOConnection. Here System>Data.ADO is the namespace and System.Data.ADO is namespace.Thus we are creating the object s by the statement:

```
System.Data.ADO.ADOConnection s;
```

since the output has been shown as C-Sharp, it is clear that the constructor has thrown no exception, as otherwise catch block would have been executed.

17.28 Accessing an Access Database using ADO

Tools Used Visual C# .NET

This program shows how access an access database using ADO. I have an Access 2000 database in project\bin\debug directory. My database name is mcTest.MDB. You can create your own database and copy in your project's debug directory. I am not attaching database with this code because of size problem.

I have a table 'Developer' in this database which has four columns. I am accessing only two columns Name and Address in this code.

Listing 17.4

```
using System;
using System.Data;
using System.Data.ADO;
public class DataAceessClass { public static void Main () {
   string strConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data
  Source=mcTest.MDB";
   string strRec = "SELECT * FROM Developer";

   // create Objects of ADOConnection and ADOCommand
   ADOConnection myConn = new ADOConnection(strConn);
   ADOCommand myCmd = new ADOCommand( strRec, myConn );
   ADODataReader outData = null;
   try
   {
    myConn.Open();
    myCmd.Execute( out outData );
     if ( outData != null )
     {
        while (outData.Read() )
        {
   Console.WriteLine("Developer Name:{0}, Address:{1}",
   outData["Name"] , outData["Address"]);
         }
     }
  }
  catch (Exception e)
  {
  Console.WriteLine("Oooops. I did it again:\n{0}", e.Message);
  }
  finally
  {
     if ( myConn.State == DBObjectState.Open )
```

```
        {
        myConn.Close();
        }
        if( outData != null )
        {
         outData = null;
       }
    }
} }
```

17.29 ADO+ : The Evolution of the Data Species

Summary: This article discusses the increased interoperability and scalability provided by ADO+, the latest incarnation of ADO.

From the beginning, software applications have been developed to access some sort of database. Distributed and Web-based applications are no exception. In a distributed scenario, however, things are a bit more complicated due to the possibility of different hardware and software platforms or object models. Despite all this, data is just data and needs to be exchanged and processed just about everywhere.

Before we entered the age of the programmable Web—the third phase of the Internet—data access was a relative issue; the primary concern was choosing the most cost-effective database server. All modules of any system had to comply with the requirements of the database server—an omnipotent entity taking full control of the entire system. Client/server applications have been the typical expression of this model.

More recently, n-tier Microsoftr Windowsr DNA systems focused on techniques to quickly and reliably access just about any sort of data: relational, non-relational, hierarchical, semi-structured, dispersed, volatile, and so forth. This unified approach to data access became the Universal Data Access strategy—the inspiring principle for the OLE DB architecture. Microsoft ActiveXr Data Objects (ADO) then materialized to complete an important mission: ferrying thousands of Windows developers from an antiquated client/server model to the wonderful world of data access components and OLE DB.

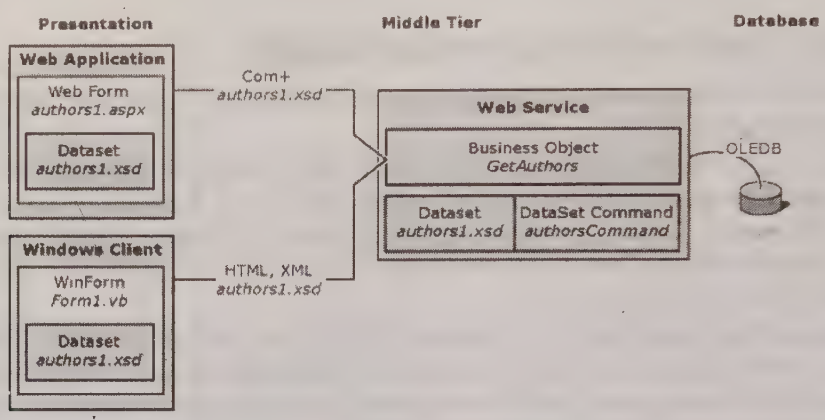
In the Windows DNA model, you have middle-tier components that get and exchange data using a common format that the OLE DB specification kindly defined for us. This format is based on the rowset format and, more often than not, is converted into a higher-level structure such as the ADO recordset.

There are pros and cons to using ADO recordsets. On one hand, they provide a rich and attractive programming interface. On the other hand, they are strictly COM-based and rather unusable in a distributed and heterogeneous environment where more platforms are involved, non-Windows platforms in particular. Interoperability and scalability are two strong requirements for modern Web systems; interoperability and a more scalable data access architecture are also the key changes in the newest ADO incarnation: ADO+.

17.30 Creating a Distributed Application

In this unit, you will create a multitiered distributed application.The application will consist of three logical tiers: Data, Business Objects, and User Interface. The data is a database in SQL Server. The business objects will handle accessing the data and distributing it to the clients. The user interface will consist of both a Web-based application and a traditional rich Windows application.

Architecture of the distributed application you will create the following:



The application you will build is a simple data look-up application. You will build both a Windows and Web-based client to display the Authors table from the SQL Server sample Pubs database. For the Web portion, you will use the Web Forms Designer to create HTML that is compatible with a standard HTML 3.2 browser. On the server, the Web Forms code will call a Web Service to retrieve a dataset that contains the requested Authors information from the database. For the Windows portion, you will build a Windows Application, which will talk to this same Web Service to retrieve a dataset containing the author information. The Windows Application will look like any Windows application (launched from the desktop, with menus, and so on) but will communicate to the Web Service using HTTP and XML.

17.31 Process for Creating a Distributed Application

Typically, you create distributed applications one tier at a time, beginning with the data, then the middle-tier business objects, and finally the user interface. The data has already been generated and is available in the Pubs database in SQL Server. Therefore, the walkthrough will start with creating the business object—the Web Service—followed by building the two user interfaces—a Web Forms page and a Windows Form.

The process for this walkthrough is as follows:

1. Create the middle-tier business object
   - Create a Web Service project
   - Create a dataset and dataset command
   - Expose the dataset from your business object

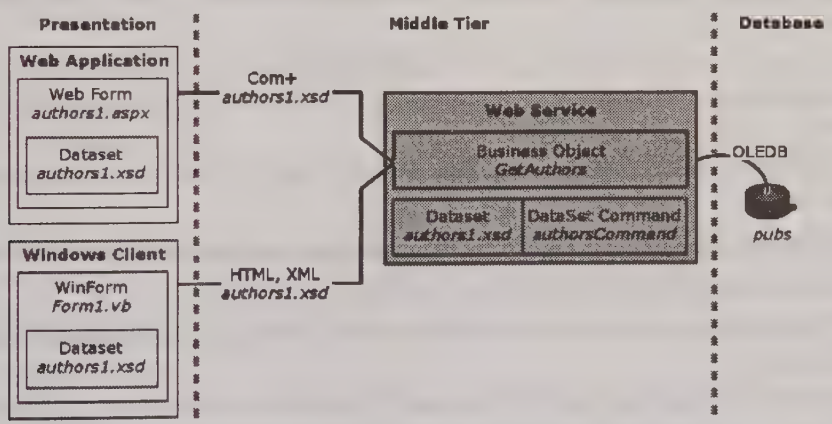
2. Create the user interface
   - Create a Windows user interface
   - Create a Web user interface

3. Deploy the solution

17.32 Create the Middle-Tier Business Object

The business object you create will run on a Web server to provide the performance and scalability necessary for a distributed application. In addition, you will expose the business object as a Web Service so that clients can use standard Internet protocols to communicate with your business object from any platform.

The Web Services project template is a Web-based project designed for creating middle-tier components that expose their interfaces as Web Services. A component will be added to the Web Service that will hold the data connections and dataset. Web Service methods will then be added to the Web Service to expose it, which will make it possible for other applications to view and modify the dataset.



To create a Web Service project

1. On the File menu, point to New, and then click Project to display the New Project dialog box.
2. Depending on what language you want to use, select C# Projects or Visual Basic.NET Projects in the Project Types pane, and then *select* Web Service.
3. Name the project AuthorsWebService.
4. In the Location box, enter the name of the Web server, http://(server name), and then *click* OK. (**Tip**: If the Web server is on your computer, you can use the server name LOCALHOST).
5. From the Project menu, select Add New Component.
6. Click Component and change the name of the component from the Component1 to AuthorsComponent in the Name text box.

In this component, you will create a connection to the data store and obtain an instance of the data using a dataset.

17.33 Create and ConFig. a DataSet

Two objects will be added to the AuthorsComponent: a ADODataSetCommand Class and ADOConnection Class. The connection creates a new connection to the database while the dataset command queries the database and feeds the data into the DataSet. The dataset command also updates the database with changes made to the DataSet by the user. Using the connection to the database, a DataSet object will be created to hold an instance of the information present in the database. The DataSet will be used later in the walkthrough to display data in the Windows Form and the Web Forms page. For details, see Introduction to Distributed Applications and Data Integration.

To create a DataSet object:

1. Click Server Explorer.

(Note: By default, Server Explorer appears as a vertical tab on the left of the screen.)

2. In Server Explorer, right-click the Data Connections node and select Add Connection from the shortcut menu.
3. In the Connection tab of the Data Link Properties dialog box, enter the name of the SQL Server where the pubs database is installed. If you have SQL Server on the local machine, enter(local).
4. Enter SQL Server logon information. The user you specify must have read/write access to SQL Server.

(Tip: Often the user name for the Pubs database is sa and there is a blank password.)

5. Select the pubs database from the drop-down list.
6. Click Test Connection to validate the information you provided, and then click OK to establish the connection.

[server name].pubs.dbo appears in the Database Connections node of Server Explorer.

(Note: If the database connection failed, see your database administrator.)

7. In Server Explorer, expand the [server name].pubs.dbo node and then expand the Tables node.
8. Drag the authors table from Server Explorer onto the design surface.The authors table appears in the designer as an ADOConnection paired with an ADODataSetCommand. A connection has now been created to the database, with the transfer of information to be handled by the ADODataSetCommand.
9. From the DataClass menu select Generate DataSet.
10. In the dialog box that appears, name the DataSet Authors1.

The dataset Authors1 has now been created and holds an instance of the Authors table in the Pubs database. The schema for the Authors1 dataset can be accessed in the Authors1.xsd file, which was generated with the dataset.

11. From the DataClass menu select Generate DataSetCommand Methods.

The Code Editor appears, showing that the following methods were generated:

```
Public Sub FillDataSet(ByVal dataset As System.Data.DataSet) Public
Function UpdateDataSource(ByVal updateDataSet As System.Data.DataSet)
As System.Data.DataSet
```

12. From the File menu, choose Save 'AuthorsComponent'.

Expose the DataSet from Your Business Object

The next step in this walkthrough is to expose the DataSet you just created from your business object. This will make the DataSet available for Windows or Web applications to use.

To add the DataSet to a middle-tier business object:

1. In Solution Explorer, double-click AuthorsWebService if it is not already open in the design window.
2. From the View menu, select Code.
3. Add the following Web Service methods:

Listing 17.5

```
// C# [WebMethod]
public webservice1.Authors1 GetAuthors() {
   Authors ds = new Authors1();
   AuthorsComponent dc = new AuthorsComponent();
   dc.FillDataSet(ds);
   return ds;
}
[WebMethod]
public webservice1.Authors1 UpdateAuthors(webservice1.Authors1  ds) {
   AuthorsComponent dc = new AuthorsComponent();
   dc.UpdateDataSource(ds);
   return ds;
}
```

1. From the Build menu, choose 'Build'.
2. From the File menu, choose 'Save All'.
3. When you are prompted to save the solution file, name it 'AuthorsDistributedApp'.

In these sections, you have created a middle-tier business object that contains a dataset bound to a SQL Server database. You added code to the middle-tier AuthorsWebService Web Service that clients will access to load and update the data in the DataSet.

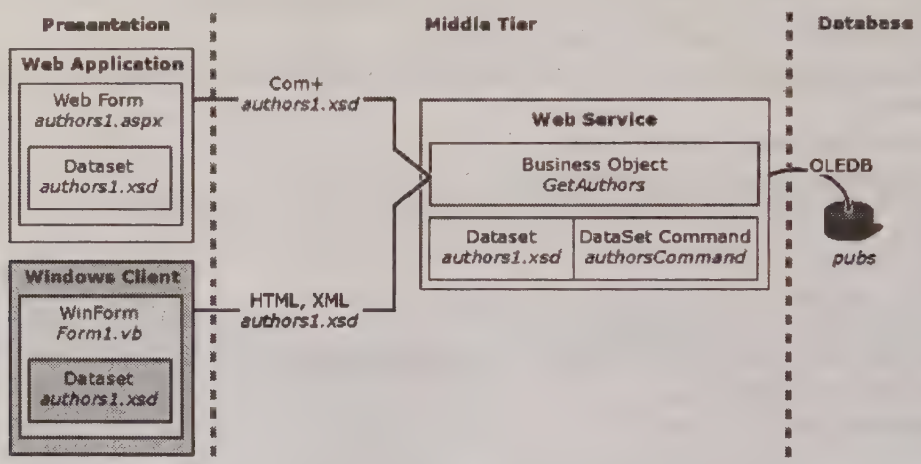
17.34 Create the User Interface

After creating a middle-tier business object for data access and exposing it as a Web Service, the next step is to create the client interfaces. There are two options in this walkthrough: a traditional rich Windows Form and a Web-based form. Both are created in this example as separate projects in the same solution.

17.35 Windows User Interface Option

A Windows interface uses the capabilities of the client machine to handle part of the application processing. Generally, a Windows interface provides greater functionality and a richer user experience than a Web-based interface. There is less of a load on the server than with a Web front end, because the server does not have to perform all of the application logic. Additionally, a Windows interface can take advantage of the resources available through the operating system, including calls to the file system and the Registry.

Windows client in the distributed application



To create a rich Windows user interface:

1. On the File menu, point to Add, and then click New Project to open the Add New Project dialog box.
2. Select Windows Application and enter AuthorsWinClient for the project name.

The AuthorsWinClient project is added to the solution. Form1 is automatically added to the project and appears in the Windows Forms Designer.

3. In Solution Explorer, right-click the form rename it AuthorsForm.
4. Add a reference to the Web Service project you created earlier:
   - In Solution Explorer, right-click the 'AuthorsWinClient' project, and select Add Web Reference from the shortcut menu.
   - Enter the location of the .disco file of your Web Service project (http://servername/AuthorsWebService/AuthorsWebService.disco) in the Location box at the top of the Add Web Reference dialog box, and then press ENTER.
   - In the Components Found window, select the components you want and click Add Reference.

You can now bind to the Authors1 dataset generated in the Web Service.

5. If the form is not open in the designer, double-click it in Solution Explorer.

6. Click the Data tab of the Toolbox, and drag a DataFormWizard onto the form.

The Data Form Wizard appears.

7. Choose the Authors1 dataset from the dataset drop-down list, and then click Next.

The Available Tables window displays all of the tables contained in Authors1.

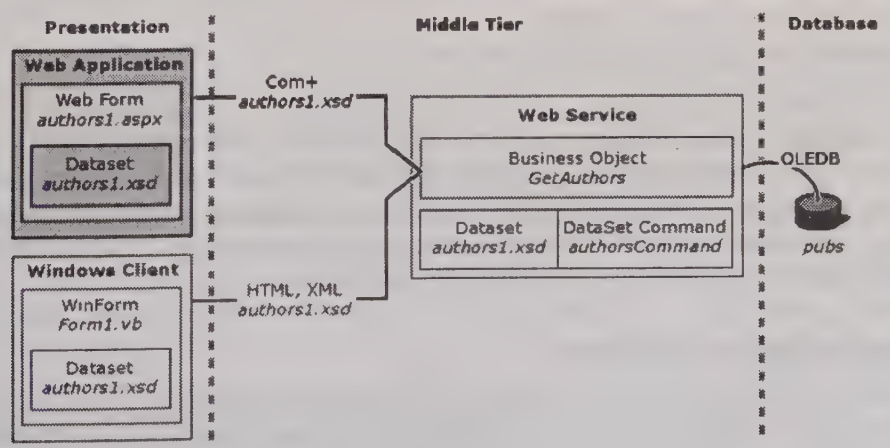
8. Select the check boxes to include update and load buttons on your form.
   - Choose the GetAuthors Web Service method from the Load drop-down list to associate it with the load button.
   - *Choose* the UpdateAuthors Web Service method from the Update drop-down list to associate it with the update button.
9. Select the All Records option to display all of the records in the authors table as a grid.
10. From the File menu, choose Save All.
11. From the Build menu, choose Build.
12. Press F5 to run the application. A window is displayed that contains an empty table with headers from the Authors table in the pubs database.
13. Click Load to populate the table, and then click Update to save the changes you make.

In this section, you added a Windows Forms project to your solution to act as a Windows interface. You connected the Windows Form to the Web Service you created in the first section and added Button controls to load and update the data.

17.36 Web User Interface Option

A Web interface helps your application reach a broad variety of client machines and browsers. All of the user-interface processing is accomplished on the Web server rather than the client. Using a Web Forms page, you will create a Web interface that accesses the same middle-tier business object as the Windows interface that you created in the previous section.

Web Forms in the distributed application



To create a Web Forms page:

1. On the File menu, point to Add, and then click New Project.
2. In the Add New Project dialog box, select C# Projects or Visual Basic Projects in the Project Types pane, then click Web Application.
3. Name the project AuthorsWebClient.

A new project is added to Solution Explorer. A Web Forms page called WebForm1.aspx is added to the project and is loaded into the designer.

4. Close the new .aspx file.
5. In Solution Explorer, rename the .aspx file to AuthorsWebForm.aspx.
6. Click Yes in the Link Repair window when asked if you want to update links.
7. Add a reference to the Web Service project:
   - In Solution Explorer, right-click the AuthorsWinClient project, and select Add Web Reference from the shortcut menu.
   - Enter the location of the .disco file in your Web Service project (http://servername/AuthorsWebService/AuthorsWebService.disco) and press ENTER.
   - In the Components Found window, select the components you want and click Add Reference.

You can now bind to the DataSet generated in the Web Service.

8. Double-click AuthorsWebForm.aspx in Solution Explorer to open it in the designer.
9. Click the Data tab of the Toolbox, and drag a DataFormWizard onto the form.
10. Choose the Authors1 dataset from the DataSet drop-down list and then click Next.
11. Choose the GetAuthors Web Service method from the drop-down list to associate it with the Load button, then click Next.
12. In the Column window, choose the columns that you want to be shown from the Authors table.
13. Select All Records to display all of the records in the authors table as a grid.
14. From the File menu, choose Save All.
15. From the Build menu, choose Build.
16. Press F5 to run the application.

A window is displayed that contains an empty table with headers from the Authors table in the pubs database. Click Load to populate the table and Update to save changes you make.

In this section, you added a Web Application project to your solution to act as a browser interface to your data. You connected the Web Forms page to the Web Service you created in the first section and added code to load the data when the form opens.

17.37 Deploying the Solution

The next step in the process would be to deploy your Web application to a server and to create an installer for your Windows application. For information about deploying your Web application see Walkthrough:Deploying a Web Solution. For information about deploying your Windows application see Walkthrough: Creating an Installer for your Application.

17.38 A Common Data Manipulation Language

Typically, today's distributed systems match the architecture(see Fig. below)

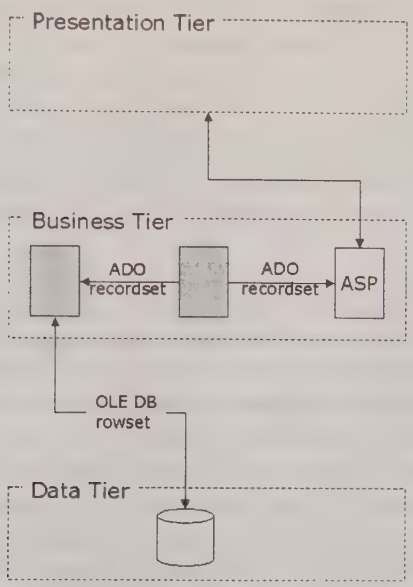


Fig. 17.6 Typical architecture for todays distributed Web systems

The presentation layer is normally based on HTML 3.2 output and works fine with any reasonably recent browser. The page is built on the Web server using Active Server Pages (ASP) and only in rather unique cases attempts to accommodate the real capabilities of the browser in terms of COM, Dynamic HTML, and XML support.

The crucial point is the middle tier, where there are usually one or more layers of business objects getting and exchanging data in response to the user's input. These components may need to pass data to each other, and in doing so they need a common data format that is easy to use, powerful, and understood by all. The ADO recordset—the ADO representation of a table or a view—is a pretty good solution, with several pros and just one big con.

ADO recordsets are flexible enough to allow you to navigate the records with extreme ease, and to apply filters and bookmarks. They also provide sorting, automatic pagination and persistence, and can work when disconnected from the data source. Recordsets can be marshalled across the tiers rather efficiently due to their native and extremely compact binary format—the Advanced Data Table Gram (ADTG) format.

The transmission of a disconnected recordset from component to component involves COM marshalling and forces the receiving component to accommodate it. In other words, only COM objects can use ADO recordsets. This is a nonissue in a homogeneous architecture where COM/DCOM reign over the business tier. Clearly, this results in great inconvenience when the

map of the involved server-side components touches on heterogeneous nodes such as mainframes or Unix platforms.

The so-called third phase of the Internet brings this scenario even closer. It preFig.s a world in which functionality is made available through Web Services—that is, satellite systems circling around us and available through HTTP. A scenario in which you will need to pass a number of records to one of these services for further processing is not that far-fetched. Nothing's easier than the Web Service—whatever its internal architecture or public programming interface—it doesn't accept ADO recordsets.

ADO, as it exists today, and recordsets in particular are great tools for manipulating data in a Windows and COM-based scenario. Unfortunately, they progressively lose their appeal as soon as your system evolves in the direction of total Internet interoperability.

17.39 The Data Species

In a perfect world, data would be accessible in the same way, and with the same flexibility, on any platform or device. Each platform or device, then, would be free to manipulate the data as needed. If you expose data through ADO recordsets you expose yourself and your application to the concrete risk of limited interoperability.

Today, if you access data through ADO and want to transmit it to remote components, either you use the same ADO recordsets you got from the data access module or you convert it into something else that can be transmitted over the network and, more importantly, understood at its final destination. As mentioned earlier, recordsets require COM marshalling and, for example, COM calls cannot always penetrate corporate firewalls. In addition, when marshalling ADO recordsets a significant amount of the overall processing time goes toward accomplishing all the necessary type conversions. In fact, you must ensure that all the values in the records map to valid data types that COM recognizes and knows how to handle.

The COM marshalling factor becomes even more significant when the involved components are physically separated and run on different machines. As a result, while moving to a totally Internet-connected world, today's Windows data species with ADO recordsets is forced to evolve in order to survive.

Want an example? In my Cutting Edge columns in the January 2000 issue of MIND and the March 2000 issue of MSDN Magazine I illustrated the use of Remote Scripting (RS) as a cross-browser technique to get data from a remote ASP page without actually navigating to it. When you invoke this way a remote function on a page, the RS infrastructure provides for marshalling back to the client what the function returns. In most cases, you need to return a recordset. However, for security reasons RS doesn't even attempt to process an ADO recordset along with any other COM object. Thus, if you use RS you must avoid using ADO recordsets and instead use arrays or strings to pass the requested information. In the March 2000 column I worked around this intrinsic limitation by building a JavaScript object from the recordset. The ASP page gets the data through ADO, and prior to the return to the caller transforms it into a JavaScript object:

```
rst = new ActiveXObject("ADODB.Recordset");
```

```
rst.open("select * from employees", "DSN=Northwind");
oRS = new Recordset(rst);
rst.close();
return oRS;
```

The client page receives this object through the RS run-time and can process it as needed with much more ease than with a string or an array. For more information and source code, see my March 2000 Cutting Edge column.

Is there a lesson to learn from this? For data interoperability that goes beyond the boundaries of a single client or server-side platform, we need an evolutionary change to the ADO model.

We need a change because the interaction of cross-platform modules requires a universal data model. Furthermore, we don't want this change to be too revolutionary because there are several good features within ADO that would be a pity to renounce.

The JavaScript Recordset object, in its inherent simplicity, is just a sign of the times. We need to extract the essence of ADO recordsets and rebuild it into something else that could be easily transmitted and processed on any platform. HTTP provides the universally accepted network plumbing. XML brings together the universally accepted infrastructure for data description. ADO+ is the evolutionary improvement to ADO that makes it a standards-based programming model for creating distributed and data-sharing applications.

17.40 ADO Plus

At the highest level of abstraction, ADO+ is ADO made more scalable and interoperable. In terms of the object model and programming issues, ADO and ADO+ are completely different entities. Nevertheless, ADO+ is born from the origins of ADO and retains its inspiring principles.

The key to enriching ADO with increased interoperability and scalability is the notion of disconnection. Disconnected recordsets are introduced in ADO 2.0 as a way to preserve system resources while allowing the client-side script code to implement interesting and interactive functionality, such as sorting, filtering, and document/view models. ADO+ has three major advantages over ADO:interoperability, increased scalability, and strong typing. In addition, ADO+ facilitates data sharing across data components and navigation through the records of a table.

From the start ADO+ has been designed to work with disconnected data sets. Disconnected sets of records can only benefit the application because they are local views of the data that are faster to process and transmit. ADO+ utilizes XML as the universal transmission format. This guarantees interoperability as long as the receiving component runs on a platform where an XML parser is available and provides a significant number of features. When the transmission occurs through XML it is no longer necessary that the receiver be a COM object. XML is a simple but powerful text-based standard that's recently gained broad industry acceptance, so now it's reasonable to expect to find an XML parser on just about every platform. Given this, the receiving component has no architectural restrictions whatsoever. Any pair of software components can share ADO+ data, as long as they agree to use the same XML schema for the format of the transmitted data. Fig. 2 illustrates how ADO+ fits within existing architectures.

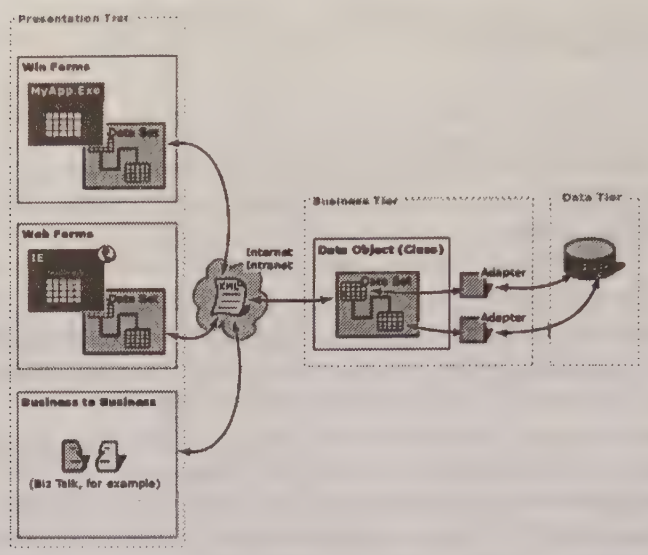


Fig. 17.7: Using ADO+ in a DNA scenario

The presentation tier can consume or create data sets through Win Forms or Web Forms—both programming elements are specific to the Microsoft Visual Studio.NET platform. In either case, the components running on the business tier receive an XML stream to process at will. These components can either rebuild an ADO+ environment or process the data as raw XML and adapt it to be sent to the physical data store. Business components can get ADO+ data, convert it to XML, and send it back to the client. Any application that can read XML can intervene in this schema at any point.

Scalability is the ability for a distributed system to serve a growing number of clients without losing efficiency. Scalability is a term that is often associated with databases because distributed systems heavily rely on databases. Scalability is a viable issue both for database servers and any software artifact that encapsulates data access. The worst enemy of scalability is the abuse of critical resources. In distributed systems, more often than not, critical means limited. An example of a critical resource that can affect scalability is database connections. The availability of a fixed number of connections poses a physical limitation to the growth of the system and its capabilities. Good software is aware of this and takes proper measures to limit the impact. How does ADO+ cope with this better than ADO? It's not so much that ADO is not scalable, but the intrinsic use of disconnected and in-memory tables that characterize ADO+ is an inherently more scalable solution. It's more scalable because a disconnected data set doesn't retain locks or keep connections open for a long time. With ADO, you have to code to obtain this; you get it for free with ADO+.

The use of XML as the transportation language for data sets promotes more effective data sharing than with ADO. First off, bypassing the COM marshalling required for COM objects such as recordsets enables components to use whatever set of data types they want. This also results in better performance because data type conversions to ensure the records comply with

standard COM data types are no longer needed. The use of XML and HTTP also allows data to jump over firewalls. Normally, firewalls allow HTTP packets to pass but block anything else that attempts to come in through ports other than 80.

17.41 ADO+ Constituent Elements

Now we will discuss the principal ADO+ objects, starting with the root of everything— the DataSet object.

Data sets are the evolution of ADO recordsets. A data set is simply a disconnected and in-memory view of the database. There is no strict relationship between a data set and a data source in the sense that you can have data sets dynamically created and tables of data without any byte of information coming from a database management system (DBMS). In other words, data sets utilize a few features that were added to ADO recordsets.

The capability of working disconnected and being programmatically created and populated with any sort of data. A data set can contain any number of tables, each of which often, but not always, corresponds to a database table or view. A table, to which corresponds a DataTable object, is simply a collection of rows and columns. Each row retains its original state together with the current state.

ADO recordsets have often been presented as a kind of general-purpose data type acting as a sort of super array. Likewise, an ADO+ data set is a sort of super recordset that provides:

1. A better and richer programming interface to host data.
2. An object model that offers a more generalized view of the data without any intrinsic binding with the actual data source.
3. An XML-based standard console for input and output.
4. ADO recordsets are basically a (possibly hierarchical) collection of records with specific methods for doing a number of interesting things, including scrolling, sorting, filtering, bookmarking, and so on. Recordsets are essentially in-memory copies of a table. A data set, instead, looks more like an SQL
5. Server or Access database that is an entity including more tables, particular views, and foreign-key relations.

The data set object model reflects this larger horizon and allows you to see the available data spanned across collections of data tables, relations, extended properties, and rows—which can also be used to describe nonpersistent data or data that is simply coming from another persistent storage medium other than a database.

ADO recordsets allow you to save content to XML and rebuild it from an external XML file. However, the default XML schema employed was optimized for exporting/importing ADO recordsets, not for real data exchange. The ADO XML schema contains a lot of information about column type and position and other metadata. This information is crucial if you want to rebuild a recordset from an XML document, but it is rather useless if you just want to pass data and make the receiver to use it. With ADO+ data sets the XML schema is leaner because the object being represented has no direct relationship with a relational database, although it can be used to closely render it. ADO+ data sets describe data, while ADO recordsets describe a collection of records taken from a table. That's the beauty of the ADO+ data model.

The ADO+ XML schema simply renders the tables and the relations defined in the data set, leaving the ADO+ run-time responsible for building a DataSet object. The ADO XML format is too specific to be truly interoperable. It is also too verbose to be efficiently marshalled without resorting to manual intervention for compressing data. There's an underlying similarity between the inspiring principle of the ADO+ data sets architecture and what led me to create the JavaScript's Recordset object. In both cases, you transform a data description (for example, an XML string) into a working object specific to the platform (that is, a JavaScript object). In doing so, you safely carry your data from tier to tier, regardless of the operating system or the software installed.

17.42 ADO+ Commands

While the DataSet object provides a tool for in-memory data storage, you need another tool tc create and initialize the various tables. This tool is the DataSetCommand object and represents a centralized console that hides the details of working with connections and commands. The DataSetCommand object allows for the retrieval and saving of data between a DataSet object and the source data store. It is responsible for pulling out data from the physical store and pushing it into data tables and relations. The DataSetCommand object is also responsible for transmitting any update, insertion, or deletion to the physical database.The DataSetCommand object accomplishes this through either SQL commands or command strings that the underlying OLE DB provider can understand. The DataSetCommand object exists in two following forms:

- SQLDataSetCommand objects; and
- ADODataSetCommand objects

These two forms of the DataSetCommand object can be thought of as proxies that mediate between a data table and a corresponding table in the data source. The data source is SQL Server 7.0 (or later) for SQLDataSetCommand objects and any other OLE DB provider for ADODataSetCommand objects.

In general, ADO+ recognizes and handles two types of data sources: SQL Server 7.0 (and later) and any data source accessible through OLE DB providers. These are also known as managed providers.

As an alternative to using DataSetCommand objects (as shown in Listing 17.7), you can work directly with managed providers using more familiar objects like connections and commands.

Listing 17.7

```
Dim oDS as DataSet Dim oCMD as SQLDataSetCommand
oDS = New DataSet oCMD = new SQLDataSetCommand("Select * from employees",
strConn) oCMD.FillDataSet(oDS, "EmployeesList")
Dim oRow as DataRow For Each oRow in oDS.Tables(0).Rows
   Console.WriteLine(oRow(0).ToString())
Next
```

Looking at this sort of ADO+ code, we will observe that it is not that much different from the old-style ADO programming. New objects, a new programming interface, new and improved features, but the same code pattern. This is shown in Listing 17.7.

Listing 17.8

```
Dim oCN As SQLConnection Dim oCMD As SQLCommand Dim oDR As ADODataReader
oCN = New SQLConnection(strNWind) oCMD = New SQLCommand("MyStoredProc
'Davolio'", oCN) oCMD.CommandType = CommandType.StoredProcedure Try
   oCN.Open()
   oCMD.Execute(oDR)
   While oDR.Read
    Console.WriteLine(oDR("LastName").ToString))
   End While
Catch e As Exception
    Console.WriteLine(e.ToString)
Finally,
       oDR.Close()
       oCN.Close()
End Try
```

Notice that the preceding code features some syntax elements introduced with Microsoft Visual Basic 7.0_in particular, the Try-Catch construct that finally retires the On Error mechanism. ADO+ code takes full advantage of the .NET run-time services that provide system objects such as the Console object.

Is there any special feature that can help you decide whether the innovative DataSetCommand approach is preferable to using traditional ADO? Using a DataSetCommand object to communicate with the data source is largely the same as using ADO connections and commands, with an important corollary: All ADO+ objects, like the DataSetCommand object, can be customized by deriving new, more specialized classes. Thus, for example, you can control how the changes to the data set are transmitted to the database and, say, optimize the performance, compress or encrypt data, perform data validation, and much more.

17.43 Strongly Typed Programming Language

With ADO you work with database objects and collections like the Recordset and Fields. There's nothing wrong with this approach, except that the actual data is perceived as a sort of parameter:

```
Set oRS = New ADODB.Recordset oRS.Open strSQL, strConn While Not oRS.EOF
MsgBox oRS("FirstName") & " " & oRS("LastName") oRS.MoveNext Wend
```

In the preceding ADO code snippet, the first and the last name are used as entry points into the Recordset and the Fields collections. Due to the Visual Basic default properties, we can write code that's relatively easy to read. According to ADO, the 'full object model path' to the a given record field would be:

```
oRS.Fields("firstname").value
```

In addition, ADO always utilizes Variant data types to comply with script-based environments (ASP in particular).

ADO+ takes database programming one step further by providing the ability to access the data stored in data sets through their natural names. To address a table in a data set you can use the ADO+ name of the table and access columns and rows using their names instead of using collection-based methods. Moreover, all variables involved have a specific (as opposite to Variant) data type. This is called strongly typed programming. The code shown above, which manipulates FirstName and LastName from an Employees table, could be rewritten like this:

```
MsgBox Employees.FirstName & " " & Employees.LastName
```

A typed DataSet is a class that inherits from DataSet. Adopting the strongly typed version of the data set is helpful because the IDE itself can inform you in real time about possible type mismatch errors, and the Microsoft IntelliSenser technology can work side by side with you to suggest methods and properties.

17.44 Speciality of ADO+

ADO+ is not just another object model for working with data. In essence, it is ADO revisited on a wider-range, cross-platform scenario. ADO+ is ADO extended to work in cooperation with any software component, on any platform, that understands XML. It models data, not databases or data providers. It pulls the data out of the database infrastructure and into a neutral environment where data is just data, no matter the storage medium, the physical organization, or the API used to access it. The key to this is XML, which represents a sort of I/O console for ADO+ data sets. Another important aspect is the disconnected and in-memory nature of the principal object, the data set.

Consequently, you can send an XML description of the data and have ADO+ server-side components build it up as a data set. Alternatively, you may have non-ADO+ components that simply take the XML and run their own parser on it, obtaining the same results. ADO+ is Windows-specific but can exchange data with just about any platform. Stay tuned for future articles following the evolution of ADO+.

17.45 How to Import ADO

TLBIMP.EXE is one of the free system utilities you get with the .NET Framework. It generates an assembly containing regular .NET metadata based on the content of the specified COM-type library. (The presence of a type library is a necessary condition for this to happen.)

Thus, to import the ADO object model into .NET you need to do as follows:

```
tlbimp.exe msado15.dll
```

The whole ADO hierarchy of objects is available in msado15.dll, which is normally installed in your c:\program files\common files\system\ado folder. In this case, TLBIMP creates a file called adodb.dll in the current folder. Unless we specify an output file name, the utility takes the name of the library. You can set an output name using the /out:name switch on the command line.

If we are using the Visual Studio.NET for building .NET applications, we do not even need to know about TLBIMP. In this case, it is sufficient that we right-click on the Reference node in the Solution Explorer and pick-up the ADO Library from the supplied list of registered COM components.

Once we do this, ADO objects are available to .NET code as native classes. The typical Visual Basic application that fetches some records out of a SQL Server table and displays them through a listview can now be rewritten with C# like this:

```
String conn = "PROVIDER=SQLOLEDB;INITIAL CATALOG=Northwind;" +
        "SERVER=localhost;UID=sa;PWD=;";
String cmd = "SELECT firstname, lastname FROM Employees";
ADODB.Recordset adoRS = new ADODB.Recordset(); adoRS.Open(cmd, conn,
ADODB.CursorTypeEnum.adOpenForwardOnly,
    ADODB.LockTypeEnum.adLockReadOnly, 0);
while (!adoRS.EOF) {
   listView1.ListItems.Add(adoRS.Fields["lastname"].Value +
     ", " +
     adoRS.Fields["firstname"].Value);
     adoRS.MoveNext();
}
```

The key thing that is going on here is what happens when the code requires a new instance of the ADODB.Recordset object. The Windows Forms application you see below is a .NET client that seemingly creates an instance of the ADO's Recordset COM object. What happens under the hood, instead, is slightly different.

The client still talks to a regular .NET class, called the Runtime Callable Wrapper (RCW), which behaves like the original Recordset object. When the RCW class is created, it immediately instantiates the target component. Then the RCW class behaves like a sort of proxy for the unmanaged component.

There is a one-to-one relationship between the unmanaged object and its .NET companion class. Different instances of COM components are managed through different RCW classes, but no new RCW is utilized for additional references to the same instance.

The RCW infrastructure takes care of marshaling method arguments and return values between managed and unmanaged code. Of course, this type of conversion takes place only when the representation of the data is different in COM and .NET. For example, BSTR variables must become String variables and vice versa.

COM and .NET differ on a number of points—exception handling, memory management, function identification, threading, to name just a few. All these differences must be hidden by the RCW layer in such a way that each end of the connection—RCW class and COM object—sees the programming interface and environment it knows how to work with.

COM clients are supposed to directly control the lifecycle of the objects they instantiate. By contrast, .NET clients know that the system garbage collector will be cleaning up memory after them. The RCW class, which TLBIMP hard-codes in the assembly it generates, inherits from base classes that know how to take care of this difference.

Likewise, .NET applications are expected to expose shared resources in a thread-safe fashion. COM clients, instead, should be aware of the threading model the server component supports. Once again, the RCW class is responsible for instantiating the target COM object within the proper type of apartment.

COM objects return success or failure through HRESULT return codes, whereas .NET clients understand from the exceptions thrown that something went wrong with their calls. RCW generously transforms any non-success HRESULT value they run across into .NET exceptions. (In light of this, COM objects returning informative content through HRESULT should be avoided.)

It should be clear by now that calling ADO objects from .NET clients is certainly doable and not particularly hard. However, each time you call into a COM server object from a .NET context, there's a lot of code that spookily slips in between your application and the actual object. You talk to an invisible proxy that, in turn, interrogates the COM object overcoming, and shielding you from all the structural differences between COM and .NET.

The proxy gives you the illusion of quick and easy communication. In practice, every time you invoke it, a bridge is built over rather troubled waters. Calculating the real weight of this procedure is completely up to you and depends on your project.

This overhead affects only the channel of communication where requests and responses travel between the .NET client and the COM component. If the COM component, in turn, calls into other COM objects—for example, a Visual Basic data access component that uses ADO or an ATL COM consumer invoking an OLE DB provider—then no further processing is required.

The overhead of the COM Interoperability layer is due to the building of a bridge between the .NET island and the mainland of COM. So, you pay the overhead price only once. A .NET client accessing ADO performs the same actions regardless of whether it issues direct calls to the ADO object model or down to a distinct data access component.

17.46 Working with Recordsets in .NET

If you use ADO, you end up working with Recordset objects. In ADO.NET, though, there is no such object. This isn't an issue per se, as long as you know what you are doing—namely, sticking to a sort of obsolete programming model.

The Recordset object has grown a lot over time. Originally, it was just the COM counterpart of the ODBC resultset dressed up with some extremely useful facilities, such as bookmarking, filtering, and sorting. As of ADO 2.6, the Recordset supports streams, XML persistence, batch update, data disconnection, manual fabrication, data shaping and more.

In spite of this wealth of functionality, the Recordset does not provide facilities to handle multiple tables of data, or using stored procedures and custom commands to batch update a data source. In addition, a Recordset is a COM object that can hardly be exchanged with modules running on other non-Windows platforms. Furthermore, if we want to exchange Recordsets with distinct modules running on other Windows-based platforms, we have to undergo the overhead of the COM marshaling and type conversion layer.

To enhance the interoperability and scalability of the whole data handling subsystem, .NET pushes a new data-centric vision of the data, opposite to the standard database-centric model of ADO and OLE DB. This manifests through three key changes which are as follows:

- Splitting the functionality of ADO Recordset in distinct and simpler objects;
- Introducing a general-purpose data container object—the DataSet;and
- Embedding in the DataSet object an XML representation of the data, basically the merging of two programming interfaces (relational and hierarchical).

To paraphrase Albert Einstein, .NET tries to keep data access as simple (and lightweight) as possible, yet robust and powerful enough to meet the most demanding needs. The ADO.NET design of the object model is a complete turnaround when compared to ADO. Instead of a huge, rather monolithic object like the Recordset, ADO.NET makes available a bunch of simpler, more specialized, objects. 'Specialization versus Versatility' is merely another dualism that software has adapted from everyday life.

A .NET counterpart of the Recordset object can be found in the DataTable object.This makes sense even though the DataTable implements only the basic functionality of a Recordset—those that have to do with data reading and writing.

Your ADO code of today uses Recordsets that follow guidelines that are no longer valid in .NET. This means that it is difficult to integrate Recordsets with other elements of a typical .NET application. You cannot use, for example, a Recordset to populate an ASP.NET or Windows Forms datagrid.

What you can do, instead, is convert the Recordset into a .NET data container class. The following code snippet takes the Recordset built earlier and builds up an equivalent DataTable:

Listing 17.8

```
DataTable dt = new DataTable("AdaptedFromRecordset");
dt.Columns.Add ("EmployeeName", System.Type.GetType("System.String"));
while (!adoRS.EOF) {
   DataRow dr;
   dr = dt.NewRow();
   dr["EmployeeName"] = adoRS.Fields["lastname"].Value +
", " + adoRS.Fields["firstname"].Value;
   dt.Rows.Add (dr);
   adoRS.MoveNext();
}
dataGrid1.DataSource = dt.DefaultView;
```

This code can be easily generalized to any recordset, as the following function demonstrates:

Listing 17.9

```
DataTable RecordsetToDataTable(ADODB.Recordset adoRS, String strTable)
{
   // Assumes the recordset is open and moves on the first record
   adoRS.MoveFirst();
   DataTable dt = new DataTable(strTable);
   // Loops through the Recordset's fields
for(int i=0; i < adoRS.Fields.Count; i++)
{
    String strColName = adoRS.Fields[i].Name;
   Type t = adoRS.Fields[i].Value.GetType();
   dt.Columns.Add (strColName, t);
}
   // Loops through records and columns
```

```
    while (!adoRS.EOF) {
     DataRow dr = dt.NewRow();
     for(int i=0; i < adoRS.Fields.Count; i++)
       dr[i] = adoRS.Fields[i].Value;

     dt.Rows.Add (dr);
     adoRS.MoveNext();
  }
    // Leaves the recordset on the last record
    return dt;
  }
```

The code first scrolls the collection of Recordset fields and adds columns to a newly created DataTable object. The new column retains the name of the ADO field and is given the equivalent .NET type.

```
Type t = adoRS.Fields[i].Value.GetType();
```

Notice that the RCW engine has somehow modified the original interface of the Recordset Field object. It added a GetType method that returns a .NET System.Type object. There's no such method in the ADO Field object.

Next, the function loops through the records and populates the DataTable Rows collection.

You use this DataTable object like this:

```
DataTable dt = RecordsetToDataTable(adoRS, "AdaptedFromRecordset");
dataGrid1.DataSource = dt.DefaultView;
```

If you plan to build a bridge to allow existing ADO code to seamlessly flow in your .NET application, then be prepared to extend this bridge to cover the whole distance from ADO objects to similar ADO.NET objects.

Using a function to hide the extra code necessary to sew together ADO and ADO.NET objects is all but object-oriented. A much more elegant solution would be arranging a sort of AdoDataTable class which inherits from DataTable and adds two new flavors of constructor. In addition to:

```
public DataTable(); public DataTable(String);
```

It would also boast:

```
public DataTable(ADODB.Recordset); public DataTable(ADODB.Recordset,
String);
```

The former will use a default name for the table, whereas the latter will take the given name. The source code for this class looks like the following:

Listing 17.10

```
using System;
using System.Data;
public class AdoDataTable : System.Data.DataTable { public
AdoDataTable(ADODB.Recordset adoRS)
   {
        RecordsetToDataTable(adoRS, "MyNewTable");
   }
```

```
    public AdoDataTable(ADODB.Recordset adoRS, String strTableName)
    {
          RecordsetToDataTable(adoRS, strTableName);
    }

    private void RecordsetToDataTable(ADODB.Recordset adoRS, String
strTable)          {
          // Assumes the recordset is open
          adoRS.MoveFirst();
          // Loops through the Recordset's fields
          for(int i=0; i < adoRS.Fields.Count; i++)
          {
                String strColName = adoRS.Fields[i].Name;
                Type t = adoRS.Fields[i].Value.GetType();
                Columns.Add (strColName, t);
          }

          // Loops through the Recordset's records to populate the
DataTable
          while (!adoRS.EOF)    {
           DataRow dr;
           dr = NewRow();
           for(int i=0; i < adoRS.Fields.Count; i++)
            dr[i] = adoRS.Fields[i].Value;

           Rows.Add (dr);
           adoRS.MoveNext();
          }

          // Leaves the recordset on the last record
    }
}
```

The .NET application needs only to reference this class in the project and use the following, surprisingly simple, code to get the same results as before:

```
AdoDataTable adt = new AdoDataTable(adoRS, "AdaptedFromRecordset");
dataGrid1.DataSource = adt.DefaultView;
```

When writing the class, put this special comment before each method:

```
/// <summary> ///  Comment here /// </summary>
```

Visual Studio.NET will use that text to provide free IntelliSense support.

Sooner you start taking the plunge into ADO.NET, the better and more effective your move will be. As I said in my previous column, there are no half measures once you opt for .NET. Either you leave your system as is, or you rethink it from scratch, defining a mid- or long-term migration plan.

17.47 Server Cursors

As of Beta 1, there is one aspect of database programming that ADO.NET does not support directly: server cursors. If you need to use server cursors in your code, we must resort to ADO functionalities.

ADO.NET has been designed to meet most expectations. This guideline has driven the team to focus on disconnected data access and client-side cursors rather than the less scalable, but sometimes necessary, server-side cursors.

A server cursor always needs an open connection and is representative of a design approach that stems from the old days of Client/Server computing. While a number of applications where we can only use server cursors still exist, Web-based systems just don't need them.

So, what do we do if our current code uses server cursors? The first thing is to take into careful consideration the option of dropping dynamic, server-side cursors in favor of static, client-side cursors,which are the preferred and recommended cursors within the ADO.NET programming interface.

Here if you can't change your code to employ client cursors, then stick to ADO.

17.48 The ADO+ Object Model

The ADO+ object model is broadly divided into two levels: the connected layer, which consists of the classes that comprise the Managed Provider, and the disconnected layer, which is rooted in the DataSet. Let's look at the connected layer first, since this is what ADO programmers will already feel familiar with. Then we will take a close look at the DataSet.

17.49 Managed Providers

Managed Providers are a core part of the ADO+ programming model. A Managed Provider loosely maps to the concept of an OLE DB provider, but the object model looks a lot more like ADO. Here you can see one of the advantages of the .NET Framework—the ability to write language-neutral components that can be called from C++ and Visual Basicr. This means that there is no longer a dichotomy of the low-level, high-performance COM components, and the slower automation layers that can be called from Visual Basic. In the .NET Framework, the OLE DB and ADO layers could indeed be merged into one layer that features high performance, and at the same time is callable from any language.

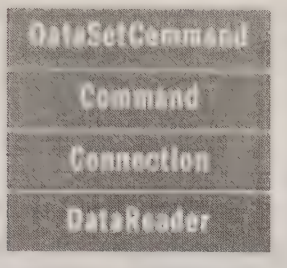


Fig. 17.8: Managed Provider

A Managed Provider consists of a Connection,a Command, a DataReader, and DataSetCommand classes. The first two should feel familiar to an ADO programmer; they are used to open a connection to a data source and execute a command against it, respectively. The DataReader loosely corresponds to a forward- only, read-only recordset. It is a highly optimized, nonbuffering, firehose-style interface for getting the results of a query executed against the data source. Finally, the DataSetCommand provides the bridge between the Managed Provider and the DataSet. The code in Fig.17.9 shows how to open a connection to a database, execute a command against it, and consume the results. Let us walk through this code. First, you will note the "Imports" and "using" directives for Visual Basic and C#, respectively. Specifying these directives enables use of the classes in the System.Data and System.Data.ADO namespaces without having to fully qualify their names (as in System.Data. ADO.ADOConnection). Next, a connection is allocated and passed the connect string in the constructor. Then, a command is allocated and passed the command string and associated connection in its constructor. Finally, a DataReader is declared. Inside the try-catch-finally block, the code opens the connection, executes the command, and returns a DataReader as an output parameter. If the DataReader returns a non-null value, the result set is read record-by-record using the DataReader.Read method, displaying the contents of the author name field.The only syntax that might look unfamiliar is the while loop on the DataReader. In Visual Basic 6.0, the code would look like the following:

```
Set myRS = myCommand.Execute Do While (Not myRS.EOF)
  Debug.Output myRS("au_fname")
  MyRS.MoveNext
Loop
```

The equivalent code in Visual Basic .NET now looks like this:

```
myDR = myCommand.Execute While (myDR.Read)
  Debug.Output myDR("au_fname")
End While
```

The main difference here is that the DataReader is not positioned on the first row by default—you have to call the Read method explicitly to get to the first record. One of the reasons for this design choice is that many people forget to put the MoveNext call at the end of their loop—so the new pattern for using the DataReader is less error-prone. Finally, note that the classes used in this example are all prefixed with "ADO" because this example uses the ADO Managed Provider. This Managed Provider sits on top of OLE DB, and can be used to communicate with any data source for which an OLE DB provider exists. The .NET Framework also ships with a Managed Provider that's built specifically for SQL Server™ —the SQL Managed Provider. Replace

all occurrences of "ADO" with "SQL" in the previous example, and you will be using the SQL Managed Provider. The SQL Managed Provider sits directly on top of TDS (the SQL Server wire format), and therefore provides significantly better performance against SQL Server than you would get going through the ADO Managed Provider and the SQL Server OLE DB provider (SQLOLEDB).

So now that there are Managed Providers, is OLE DB going away? Hardly. OLE DB is a specification that exposes data stores in a very rich way. It is a set of interfaces that covers all the features of all of the different types of data stores out there_in other words, a superset of all the store APIs. A key design goal was the ability to replace any existing store's proprietary API with OLE DB. By contrast, Managed Providers have taken the opposite approach: the lowest common set of functionality is exposed to the developer. The result is that interacting with a Managed Provider is extremely easy, and it is quite easy to create one as well; however, the functionality that it exposes is minimal. So who should write a Managed Provider? In the past, Microsoft recommended writing an OLE DB provider (or OLE DB Simple Provider) for any type of data that needed to be exposed to an application. These days, however, if it is only data that needs to be exposed. Microsoft recommends that it simply be encoded as XML—so it can plug directly into the rich set of XML services that Microsoft provides. If a true store (with rich functionality) is to be exposed, create an OLE DB provider for it. This enables the store to plug into both the .NET Framework using the ADO Managed Provider, and the many rich OLE DB consumers available (such as Microsoft Excel, Microsoft Access, and the SQL Server distributed query processor, replication engine, and Data Transformation Service). Finally, if a data store vendor has a proprietary wire format, creating a Managed Provider will result in some significant performance benefits. Since there will invariably be many people who will want to write one, a Managed Provider SDK is scheduled to ship in a future .NET beta.

17.50 The DataSet

Let us now look at the disconnected layer, the DataSet. The simplest way to describe the DataSet is as a local buffer of tables, or a collection of disconnected recordsets. One important difference between a DataSet and a collection of disconnected recordsets is that the DataSet keeps track of the relationships between the tables it contains—it is in many ways an in-memory relational store. The DataSet also exposes a rich programming model that is only possible because all data is known to be stored in a local cache. This is in contrast, the disconnected recordset, which only supports the standard MoveFirst, MoveNext, MovePrevious, MoveLast methods of the connected recordset. This difference is the key philosophical departure from ADO: instead of trying to roll many different concepts (forward-only, client, and server cursors) into one object, and thereby only being able to expose their lowest common denominator, ADO+ separates them. By making the DataSet a first-class citizen, all the underlying richness of the client cache can be exposed.

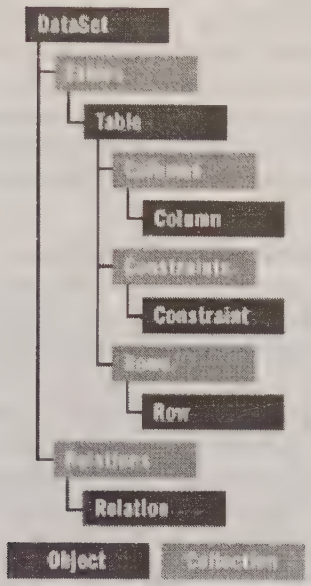


Fig. 17.9: DataSet Object Model

A DataSet contains a collection of DataTables (the Tables collection). A DataTable represents one table of in-memory data. It contains a collection of columns (the Columns collection) that represents the table's schema. A DataTable also contains a collection of rows (the Rows collection), representing the data held by the table. It remembers the original state along with the current state, tracking the kinds of changes that have occurred. In all cases, the columns are added to the DataTable using the Add method on the Columns collection, and the column is assigned a name and a datatype.

The CustomerID column is set as a unique, non-nullable column (this will be the primary key of the table), while the Rebates column is given an expression that is driven off of the Purchases column, demonstrating the DataSet expression language capabilities. Next, the primary key of the table is created from a combination of any number of columns, represented as an array of DataColumns. In this example, the primary key is just the CustomerID column. Finally, data is added to the table by calling the NewRow method on the DataTable and storing the column values in each DataRow. Note that the Visual Basic code indexes into the DataRow using the column name, while the C# code indexes using the column ordinal. Of course both forms are valid for any language.

Now let's see how to load a DataSet using a DataSetCommand, how table relationships allow for navigation and data retrieval between tables, how the Select method provides support for filtering and searching, and finally how the DataSet supports setting constraints on data.

17.51 Populating DataSets with DataSetCommands

We have seen how to load schema and data into the DataSet using the object model directly. A DataSet can also be populated using the DataSetCommand object of a Managed Provider, or by loading XML in directly. What is important to remember is that regardless of how it is loaded, the DataSet never has any knowledge of the source of the data it contains. It keeps track of any changes that are made via the object model, and exposes these changes programmatically so that a DataSetCommand can obtain them at any time and update the data source in an optimistic fashion.

Under the covers, the DataSetCommand creates a Connection object, which it assigns to its ActiveConnection property, and a Command object, which is assigned to the SelectCommand property. This, of course, can be done by coding explicitly to the Connection and Command objects as well. The DataSetCommand's FillDataSet method is then called, passing in a DataSet. Note that the DataSetCommand can create both the schema and the data inside of the DataSet. If a DataTable called Customers already existed in the DataSet, the data would simply be added to that table (and the schema of the table augmented as necessary). Also note that the connection that was created is closed after the DataSetCommand finishes loading the DataSet. This is in agreement with the disconnected nature of the operation—the connection is only open long enough for the command to be executed and the data loaded into the DataSet. In addition to the SelectCommand property, the DataSetCommand also has the InsertCommand, the UpdateCommand, and the DeleteCommand properties. These properties are used when the DataSetCommand's update method is called. At that point, the DataSetCommand opens the connection and fires off the appropriate insert/update/delete commands against the data source, corresponding to the changes that were made to data in the DataSet.

17.52 XML Integration

One of the most important design goals for ADO+ was powerful XML support. Microsoft designed ADO+ hand-in-hand with the .NET XML Framework—both are components of a single architecture. The convergence of ADO+ with the XML Framework happens in the DataSet. For starters, the DataSet has methods that can both read and write XML. For reading XML, the XML Framework parser is used, either explicitly (if an XmlReader is passed into the ReadXml method) or implicitly (via overloads that take other parameters, such as a filename). For writing XML out, the XML Framework well-formed writer (XmlWriter) is employed.

Note that regardless of where the data originated, the DataSet can save out its contents (both schema and data) as XML. The schema is encoded as an internal W3C Schema section, commonly known as XSD, and the data is encoded as XML that conforms to that schema. Because the DataSet's native serialization format is XML, it is an excellent medium for moving data between tiers in a disconnected fashion just like the disconnected recordset. Indeed, .NET Web Services make heavy use of DataSets to transport data in the context of a schema between tiers of an application. Just like populating the DataSet via its object model or through Managed Providers, loading the DataSet with XML is a two-stage process. First, the schema is created, and then the data is loaded. If the XML document comes with a schema, that schema is used to create the relational structure of the DataSet. If not, the DataSet can infer schema from the containment

relationships within the document—generally speaking, elements that are not scalar-valued are mapped to tables, whereas attributes and scalar-valued elements are mapped to columns. The process of inferring schema is useful when first constructing an application that has to consume XML that comes with no schema. But for production applications, it is highly desirable to take the inferred schema, modify it as appropriate, and load that schema in before the actual data is loaded. That way, the process of loading the document is deterministic,so you don't have to worry about what a slight change in the incoming document will do to the inference heuristics. Whatever the means for setting the DataSet's schema, when it comes time to load XML into the DataSet the following rules are used:

1. Elements with a certain name are mapped into the DataSet table of the same name.
2. Attributes and scalar-valued subelements are mapped into columns of that table.
3. The schema of the table is expanded as appropriate if the columns aren't already in the DataSet, or if the DataSet doesn't already contain a table by the same name.

When loading an XML schema into the DataSet, a table is created for each complex type in the schema, and containment is expressed using primary key/foreign key relationships. The key columns are automatically injected into the schema, and a DataRelation is added for each pair of related tables. Therefore, when loading data that resides in multiple tables from SQL Server 2000 or any other database that supports XML, it is in some ways more convenient to use the XML loading facilities rather than the DataSetCommand. The former enables loading all the data in one step, whereas the latter requires a DataSetCommand per table, and also entails setting up the relationships between the tables manually. But some data has both a hierarchical structure and a partial relational structure, and you want to be able to round-trip the data and feed it to various applications. How does this work? The XmlDataDocument, a subclass of the XmlDocument, makes this possible.

17.53 The XmlDataDocument

The process of reading the XML into the DataSet is known as *shredding*. This term originates from the fact that the XML is mapped into tables, and some of the inherent hierarchical structure and document characteristics (such as ordering and whitespace) are lost. This is appropriate for many applications, since the XML that the applications have to read originates from highly structured sources (a relational database such as SQL Server 2000 that emits XML, for example)—so there is no loss of important information from the shredding process. However, many applications will expect to read documents that have only partial relational content.

A good example is a purchase order—a hierarchical document with a relational subset (line items). Ideally, you could still treat this kind of semi-structured data as a document, but be able to view the relational content using a relational API (such as the DataSet). Moreover, after the relational content is modified (using controls bound to that DataSet, for instance), saving that document out should retain all of its original structure and fidelity. The only difference from the original document should be the new values for the nodes that correspond to the rows that were changed. This enables round-tripping business documents through the relational infrastructure, a scenario that will become increasingly important in the emerging B2B world.

The XmlDataDocument provides that same functionality in ADO+. This object is a subclass of the XmlDocument (the .NET version of the W3C-DOM), so it is a full-fidelity representation of the XML document. But the XmlDataDocument is a special kind of DOM_it can be loaded with

schema mappings, and that schema is used at load- time to determine how to map the document relationally. When the document is loaded, the relational subset can be obtained using the DataSet property on the XmlDataDocument .

The most positive thing about the relationship between the DataSet and the XmlDataDocument is the flexibility it provides. You can write a single application using a single set of data, and plug it into the entire stack of services built around the DataSet (such as Web Forms and Win Forms controls, and Visual Studio .NET designers), as well as the stack of XML services (such as XSL/T and XPath). Therefore, you do not need to choose ahead of time which set of services to target with the application—both are available.

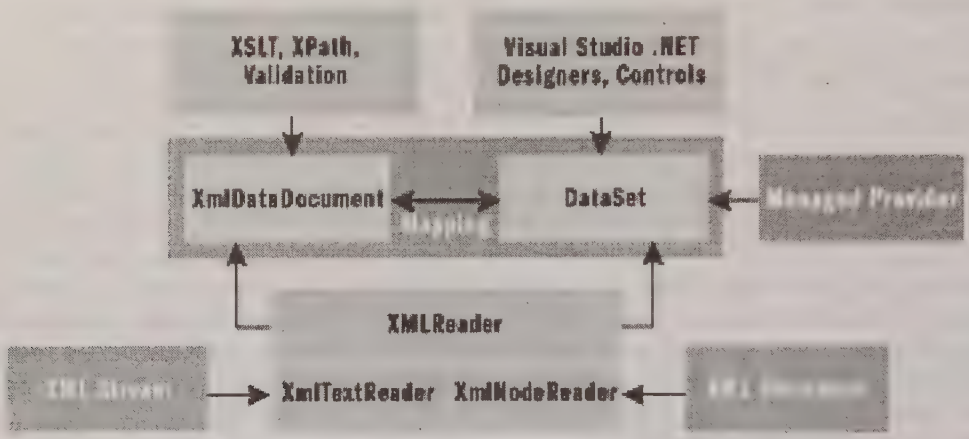


Fig. 17.11: ADO+ Architecture

For example, data that is loaded into the DataSet via a DataSetCommand can be transformed using XSL/T by creating an XmlDataDocument and passing the DataSet into its constructor. From there, the data is treated just like any other document by the XML Framework. Of course, this process works the other way, too—the relational portion of a document can be mapped into a DataSet, which can be bound to data-aware ASP+ controls, such as a DataGrid, on a Web Form. Fig. 10 depicts the complete ADO+ architecture.

17.54 Visual Studio .NET Data Designers

Visual Studio .NET will provide rich support for designing data-aware applications. The data toolbox has design-time support for many of the ADO+ classes, including the ability to browse data sources using the Server Explorer, conFig. DataSetCommands using a wizard, graphically create queries, produce typed DataSets from a database or XSD schema, produce complete master-detail forms using the Data Form Designer, and much more. The data designers expose the runtime classes directly rather than having to wrap them with their own classes, as has been the case in the past.

17.55 Typed DataSets

As mentioned in the previous section, the Visual Studio .NET data designer tools can produce typed DataSets from a database or XSD schema. This enables ADO+ programmers to refer to rows inside the DataSet as if they were strongly typed objects, a capability that ADO programmers have often requested for the recordset. Typed DataSets are subclasses of the DataSets that are generated automatically by the data designer tool based on some schema, and they have strongly typed properties that reflect the data model of the schema. For example, accessing data in a typed DataSet that contains customers ends up looking like this:

Listing 17.11

```
// in C#
   CustDataRow cust = ds.Customers["Mike"];
   String address = cust.Address;
Instead of this:
// in C#
   DataRow[] cust = ds.Tables["Customers"].Select[_
                "CustomerID = 'Mike'"];
   String address = cust[0]["Address"];
```

The advantage that typed DataSets embody is the ability to access data using a typed class metaphor, while still being able to tap into all of the services of ADO+. Only one copy of the data exists—the data stored inside the DataSet. Thus, you can view typed DataSets as an object-view mechanism on top of the DataSet.

Conclusion

ADO+ is the .NET class framework for data access. While it stays true to its ADO roots, ADO+ also represents a big step forward. The two key enhancements are first-class support for the disconnected programming model and rich XML integration. Hopefully you have gotten a taste for what the ADO+ object model feels like; for further information, refer to the .NET Framework SDK class reference. The .NET page on MSDNr at http://msdn.microsoft.com/net/ has a link to the SDK download page, where you can obtain the full pre-beta redistributable. A great place to start learning more is the ADO+ section of the How-To QuickStart? which is accessible from the main icon that is placed on the desktop after the SDK installation completes. The site also contains pointers to the .NET newsgroups where you can leave your feedback and interact with other developers interested in .NET.

Listing 17.12

```
//sharp1.cs
class xyz1 {
   public static void Main()
{
   try
{
   System.Data.ADO.ADOConnection s;
   s=new System.Data.ADO.ADOConnection();
   System.Console.WriteLine("C-Sharp");
}
```

```
   catch(System.Exception e)
{
   System.Console.WriteLine(e.ToString());
} } }
//sharp2.cs
using System.Data.ADO;
class xyz2 {
   public static void Main()
{
   try
{
   ADOConnection s;
   s=new ADOConnection("Data Source=yyy");
   s.Open();
   ADOCommand c;
   c=new ADOCommand("Create Table a1 (vno integer , name
   char=(10))",s);
   c.ExecuteNonQuery();
   System.Console.WriteLine("C-Sharp");
}
   catch(System.Exception e)
{
   System.Console.WriteLine(e. ToString());
}
}
} //sharp3.cs
using System.Data.ADO;
class xyz3 {
   public static void Main()
{
   try
{
   ADOConnection s;
   s=new ADOConnection("Data Source=yyy");
   s.Open();
   ADOCommand c;
   c=new ADOCommand("insert into a1 values(1,'C-Sharp')",s);;
   c.ExecuteNonQuery();
   System.Console.WriteLine("C-Sharp");
}
   catch(System.Exception e)
{
   System.Console.WriteLine(e.ToString());
} } }
//sharp4.cs using System.Data.ADO; class xyz4 {
   public static void Main()
{
   try
{
```

```
    ADOConnection s;
    s=new ADOConnection("Data Source=yyy");
    s.Open();
    ADOCommand c;
    c=new ADOCommand("delete from a1",s);;
    c.ExecuteNonQuery();
    System.Console.WriteLine("C-Sharp");
}
    catch(System.Exception e)
{
    System.Console.WriteLine(e.ToString());
} } }
```

17.56 Connecting to a text file using ADO.NET

LastName	FirstName	Company	Address
Smith	Fred		
Hope	Bill		
Beer	Joe	CFY	310 E. La Palmer St.
Kunda	AJ	HP	100 Birch Mill Rd

Fig 17.12 - Output to ListView from tab delimited text file

Let us learn how to use ODBC with a text file in .NET. The first step is to create a Data Source Name (DSN) in order to connect through ODBC. This can be done by bringing up the ODBC Drivers Administration tool through the control panel:

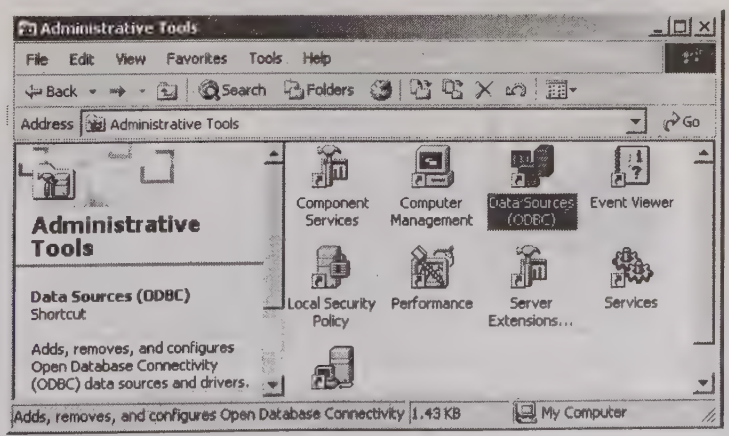


Fig 17.13 - ODBC Data Sources under the control panel

You can then choose the Add button to create a new DSN on the User DSN tab.

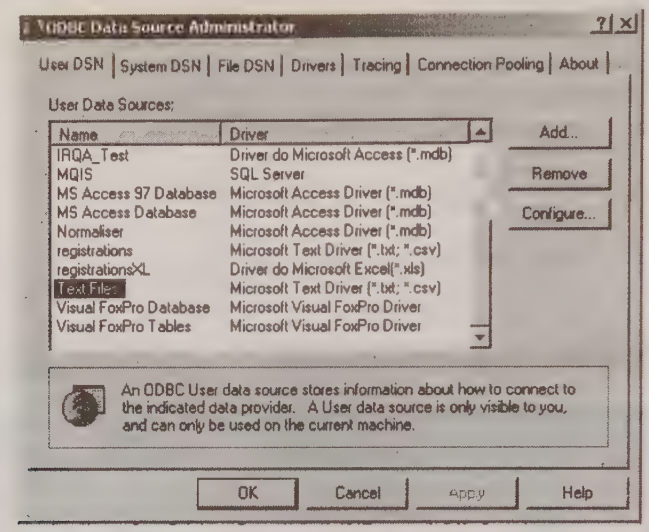


Fig 17.14 - Creating a new DSN for reading the text file

The Add Button brings up a wizard so that you can pick the driver you are interested in. In our case, it's the text driver:

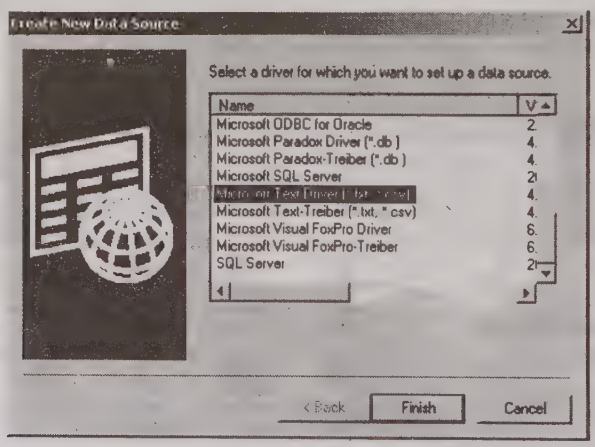


Fig 17.15 - Choosing a text ODBC driver

Choosing the text driver brings up the Text Setup Dialog. Here you can set the name of the datasource, the directory it is located in, and the extension:

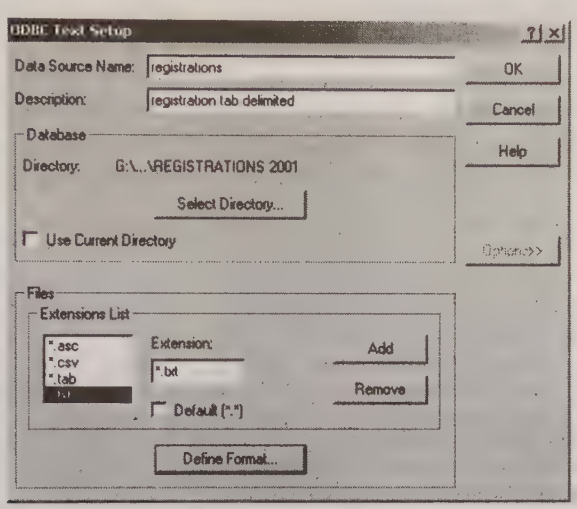


Fig 17.16 - setting up the ODBC Text Data Source for our data

Clicking on the Define Format button gives you some important additional options:

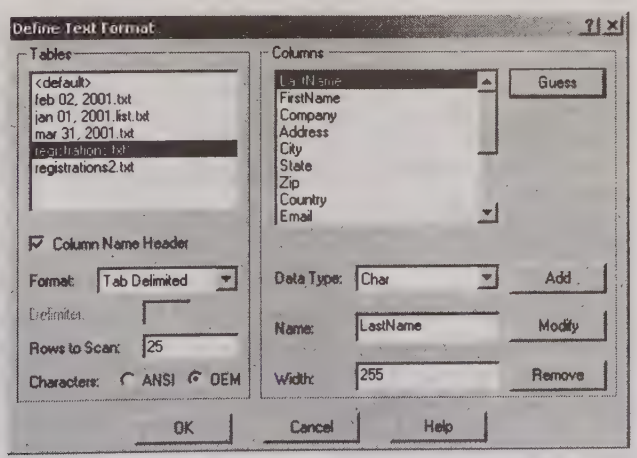


Fig 17.17 - Telling ODBC the particular format of our data

Each text file is treated as a table and if you check column name header, the names of the columns are extracted from the first line of the file. This dialog also let's you choose if the file is tab delimited, comma delimited (csv), or custom delimited. In the custom delimited format you can choose your own delimiter such as | or \. You can also set your column names in this dialog or have Microsoft guess for you from the column header line.

Once you have set up your ODBC Data Source, you are ready to use it in C#. Below is the code used to read 4 of the columns into a list view. The Data Source driver names the tables after the file names (e.g. 'registrations.txt' or 'mar31, 2001.txt'. The text database is considered the entire directory of text files. The code shows how to read data from one of the tables, registrations.txt:

Listing 17.13

```
try
 {
  // create a new ADOConnection to
//the text file through ODBC and  an existing Data Source
   ADOConnection conn = new
ADOConnection("Provider=MSDASQL;DSN=registrations;");
  //  create a DataSet Command that selects all the records from
  //the registration.txt table (which in this case is a file)
   ADODataSetCommand AdoCmd = new ADODataSetCommand("SELECT *
FROM registrations.txt", conn);
  // fill the dataset with the registration.txt table
   AdoCmd.FillDataSet(dataSet1, "registrations.txt");
   DataTable ContactTable = dataSet1.Tables[0];
   int count = 0;
   // loop through each row of the table and fill 15 rows of the
listview
   foreach (DataRow dr in ContactTable.Rows)
   {
     listView3.ListItems[count].Text = dr["LastName"].ToString();
     listView3.ListItems[count].SetSubItem(0,
dr["FirstName"].ToString());
     listView3.ListItems[count].SetSubItem(1,
dr["Company"].ToString());
     listView3.ListItems[count].SetSubItem(2,
dr["Address"].ToString());
     count++;
     if (count > 15)
      {
        break;
      }
    }
 }
catch(ADOException ae)
 {
   Console.WriteLine(ae.Message.ToString());
 }
```

This should also give you an idea of how to connect to databases through ODBC such as Oracle, Informix, Sybase, or Interbase. All you need to do is set up the appropriate Data Source through the Administration tools and use the code above to access your tables.

18

ASP.NET

18.1 Active Server Pages

In Microsoft Internet Information Server(IIS) 3.0 the server side execution that enables one to run ActiveX scripts and ActiveX server components on the server is the Microsoft Active Server Pages (ASP).Developers are able to create dynamic Web content and powerful web-based applications by combining scripts and components.

Dynamic content meáns Web pages that are customized for every user on the fly, based upon their actions or requests. For example, new visitors to your site may be served a welcome page different from that shown to a returning visitor or an on-line catalogue query may be served the latest information.

The open development environment offered by ASP supports both Microsoft Visual Basic Scripting EDition(VBScript)and JScript.This feature enables reinforcement of existing investements that use these scripting languages.

The Active Server Pages feature of IIS 3.0 requires Microsoft Windows NT Server 4.0 running IIS 2.0 or Windows NT Workstation 4.0 running Peer Web Services.

The ASP feature does not require Service Pack 2 for Windows NT 4.0, although it is recommended that all Windows NT systems receive the Service Pack 2 updates. The ASP feature contains a subset of the updates found in Service Pack 2.

Installation of ASP will upgrade IIS version 2.0 to version 3.0. The other IIS 3.0 features_Index Server 1.1, Microsoft NetShow™ , FrontPager 97 Server Extensions, and Crystal Reports_add significant functionality to IIS, but are not required to take advantage of ASP. Active Server Pages is a component of IIS 3.0, which is a free, downloadable, and integrated feature of Windows NT Server 4.0.

18.2 Features of ASP

1. A new generation of Web based applications can be developed by ASP.Such applications help in extending sales and customer service to the Web, and providing access to corporate databases and applications to any browser on an intranet.

2. Active Server Pages allows to extend the power of scripting on the server with ActiveX server components which are created using Microsoft Visual Basic, Visual C++, Java, and other languages.
3. Active Server Pages provides native support for both Microsoft, JScript and VBScript. ActiveX scripting plug-ins are available for REXX, PERL and Python.
4. Active Server Pages can work with any Web browser. The output of an ASP file is plain HTML, the content of which can be customized with regard to the capabilities of the client.
5. Active Server Pages allows defining application and session variables that can be carried across multiple pages in a Web site. This can be as simple as remembering a user's name, and it is necessary in applications such as online shopping.
6. Active Server Pages makes it easy to bring legacy database applications to the Web.
7. Active Server Pages runs on Microsoft Windows NT Server 4.0, Windows NT Workstation 4.0 with Peer Web Services, and Microsoft Windowsr 95 with Personal Web Server. Windows NT 3.51 is NOT supported. Windows NT 4.0 on MIPS is also not supported by ASP.

18.3 Active Server Pages and Security

Active Server Pages being a component of Internet Information Server, uses Windows NT Security. ASP files can be restricted to certain users through secure Windows NT authentication, basic Web authentication, or client-side certificates. Besides, all client-to-server communications can be secured with Secure Sockets Layer (SSL).

18.4 Compatibility of ASP with other Applications

An Active Server Pages application can integrate with any ODBC compliant databases including Microsoft SQL Server™ , Oracle, Sybase, Informix, and DB2 databases. Any OLE 2 application, such as Lotus Notes or Microsoft Excel, can also be scripted to access or process information. You can also write components to access online data feeds and legacy mainframes.

Active Server Pages provides all of the functionality of CGI applications in a more user-friendly and robust environment.

ASP is an easier way for your server to access information in a form not readable by the client (such as an SQL database) and then act as a gateway between the two to produce information that the client can view and use.

With CGI, the server creates as many processes as the number of client requests received. Morethe number of concurrent requests, more the number of concurrent processes created by the server. However, creating a process for every request taxes time and server RAM.Besides, this can restrict the resources available for sharing from the server application itself, leading to slow performance.

Active Server Pages runs in the same process as the Web Server, handling client requests faster and more efficiently. It is much easier to develop dynamic content and Web applications with ASP.

ISAPI applications require all of the programming and layout to be contained in a .dll file written in C++. ISAPI applications are thus more difficult to create and maintain. With ASP files, an HTML writer can script an external component and format the output. ASP separates the layout and design from the business logic.

PERL and other scripting languages are not robust development tools by themselves. Active Server Pages provides a familiar framework and objects for building complex applications that require data from relational databases and legacy sources.

ASP supports virtually any scripting language to build these applications. Third parties are currently developing additional scripting engines, such as PERL, which will be announced when ready.

Active Server Pages supports ActiveX server components written in any language, including Java. In addition, ASP includes the Microsoft Windows reference standard Java Virtual Machine.With Active Server Pages,one can bring ones existing skills and knowledge, data sources, components, and applications to the Web. Other tools create either static HTML or lock into a non-standard programming model or language.Since ASP is based upon the leading industry standards, it is easy to build, maintain, and can evolve powerful interactive Web applications.

Netscape LiveWire requires the use of JavaScript, while Active Server Pages supports the use of virtually any scripting language, with native support for VBScript and JScript. ASP supports components written in any language, while LiveWire supports only Java components.

LiveWire applications must be manually compiled after each change, and then the application stopped and restarted. Active Server Pages recognizes when an ASP file changes and automatically recompiles the application at the next request.

18.5 ASP .NET - A New Operating System

In the recent phase, computing is centered around internet with more emphasis on network than devices.For such requirements,Microsoft announced the .NET Framework It has two elements one for developing windows based applications named WinForms and the other for internet, codenamed ASP.NET.Both possessing rich features and rapid application development environment with easy to use WYSIWYG editors. Let us see what the asp.net has to offer.

First let us see what we were lacking before the .net came.

1. The internet itself is a stateless environment, i.e. when a browser requests for a page the server flushes out the required page and ends its interaction with the server and forgets about it. Hence it is not possible to maintain an interaction between server and the browser, since no values passed are stored for future reference.
2. Very little information is exchanged between the client and server machines since they both can be running a different software or even different operating systems. In other words there was no language which could universally communicate between the client and server machines.
3. ASP 3.0 is considered a little clumsy and non structured. One had to write long codes to program even basic things like validation in a WebPages. Such huge quantity of unstructured code cannot be managed.

4. Browser dependency was a big headache for web programmers.Programs needed frequent adjustments to accommodate browsers which did not have the resources to run such applications.
5. Now-a-days every application in the web has a back-end database attached to it. Most of the websites are required to connect to the database to display the data.But connecting to the database required some expertise in ADO (ActiveX Data Object), ODBC etc besides a lot of time and resources.

ASP.NET is the web development platform Microsoft built up from scratch to offer rich and greater services for web developers. Just Like ASP, ASP.NET is a server side technology for dynamically creating web pages. ASP is a scripting framework, where scripting languages such as VBScript and JavaScript are used to produce the final content.

ASP.NET being built on top of Common Language Runtime (CLR) which provides language neutrality which means that .NET applications can be written in a variety of languages such as VB.NET, C#, JScript.NET, etc. These applications are compiled into common intermediate Language (IL) irrespective of the language in which they are written enabling useres to avail features present in one language from another language.

18.6 ASP.NET and Compatibility

ASP.NET offers backward compatibility for already existing ASP applications. Already existing ASP Applications works well within ASP.NET framework, but it is strongly advisable to convert the applications to ASP.NET to exploit completely the rich services that are offered by .net platform.

At the core of asp.net is the Common Language Runtime (CLR). This is the heart of .net framework which takes care of all the activities like compiling, instantiating, safety execution, inheritance from other components etc. Developers can write codes availing the classes and functionality exposed by the compilers and other development tools in CLR. Besides it supports three languages VB, Jscript and C# (see sharp) and third party component support.Let us list down some the features that has made this technology versatile.

1. Developers using asp.net can forget about browser dependencies .ASP.Net solves the problem since it verifies with each browser about the version, capabilities etc before sending the requested page and the output accordingly.
2. Provides a set of management features that maintain the state of a page between request.These are in the form of state bag which can be programmatically deployed.
3. Provides an Event based programming model .Each form may consist of buttons-radio button, checkbox button etc. for raising an event eg. buttonOn_click .The programmer has to write event handlers keeping in mind that each event processing makes a round trip to the server.The greatest merit is that .net acts intelligently and uses the minimum resource and time to execute this.
4. User control, a new feature enables one to create controls of their own or use third party control to enhance their applications.It throws open a new market for web controls which will be a boon to IT companies.
5. ADO+ an advanced version of ADO offered by asp.net has rich features and can be easily coded.

6. Advanced caching features help the programmer to decide on what and when to cache and when to empty it.
7. An XML based web service for providing online news etc. offers a new set of features to the developers.
8. It presents an abstract and consistent model. This is much different from the unstructured code used in ASP3.0.
9. Provides a rich set of built in controls making programming in asp.net a pleasure.

Remember that these are just the features available in Beta.The final version is yet to be shipped. One can expect more features in the final version.One may say that asp.net is more like an operating system. It allows developers to write components in their own chosen language and still utilise the features of .NET by refencing the class and other functionality.

18.7 Introduction to ASP.NET

This is not simply a new version of ASP with a few new features and some bug fixes like ASP 3.0 is when compared to ASP 2.0. This is a whole new system founded on the strong points of ASP and carrying them further.

ASP.NET files runs in the NGWS Runtime. The NGWS Runtime is basically a run-time environment, which manages the execution of codes and makes programming easier. It incorporates in itself, many features that were to be programmed whenever needed. Thus the programmer is freed from worrying about codes for everything. The runtime automatically handles object references and frees them when they're no longer being used. It also manages system memory, which helps eliminate memory leaks and other traditional programming problems. Besides, the new runtime allows cross-language integration and exception handling since all the languages are running in the same environment. For example you can now derive a class in any language from a class written in any other language or pass an instance of a class from one language to another. This is made possible by the runtime and the new common type system and environment it provides, characterized by:

1. Increased performance ie;Cross-language re-use of components which leads to Better object-oriented programming support
2. Automatic memory-management
3. The ability for one to work more freely with people who use different languages

18.8 ASP+ Platform

The primary fact about ASP+ pages is that they have a variety of new extensions. A basic ASP+ page uses .aspx as the filename extension. A Web Service uses .asmx and a new construct called a 'pagelet' uses .aspc. These new filename extensions are there for the reason that, ASP+ runs side by side with the existing ASP infrastructure.Since they don't share session state, application state, or anything else, they will coexist on the same server. These new filename extensions are therefore required so that IIS can call the appropriate ISAPI filter to handle processing. Managed code is the term Microsoft uses for addressing the code that sits on top of the new .NET runtime. This code exploits the new .NET framework including garbage collection, simplified

deployment, etc. One interesting thing to note here is that except for some host-specific code, almost all of the new ASP+ is written using the .NET runtime in a new language called C#. ASP+ supports multiple hosts. The first beta will only support IIS5 on Windows 2000. However, upon release it is expected to run on IIS4 on Windows NT 4.0 as well as in IE 5.5 using a new offline feature called My Web.

On it's first request for the script, the runtime compiles the code and keeps a cached copy of the compiled result. This cached copy can then be used for subsequent requests for the script. This enchances performance because after the first request, the code can run from the much faster compiled version. The runtime watches the source file on the file system. When the original source file changes, it automatically pulls the compiled version out of the cache so that when the next request comes in it'll be re-compiled. These changes, along with other feature enhancements in ASP+, will hopefully effect a 250% increase in performance.

18.9 ASP+ Page Framework

ASP+ introduces a new page framework that provides an object model on the server for the contents of the page. This framework includes a new class of controls termed server controls. The addition of server-side comments that allow to comment out an entire block of code, data-binding expressions that allow to bind data to server controls, and new directives that allow you to declaratively set a number of page options etc.are some of new features. Server controls is perhaps the most useful new advance in the page framework. A server control is a control on the page that is instantiated on the server. Its namespace is merged with the page namespace so that one can manipulate it in code without having to alter the underlying HTML explicitly.

18.10 ASP+ Web Services

Web Services are one of the coolest new features in Microsoft's .NET platform. A Web Service allows you to take an object and use HTTP to invoke it, rather than RPC. The advantages to this are clear. Instead of tightly coupling your objects with DCOM and relying upon RPC mechanisms to communicate, the common HTTP protocol helps extend your services out to the Internet at large and take advantage of all the infrastructure support for load balancing, security, etc. that has been built into HTTP. ASP+ provides a simplified model for creating Web Services. Instead of writing an .aspx file, you write an .asmx file, which contains a class with whatever methods the developer wishes to expose. ASP+ compiles the .asmx file on demand, generating the required SDL contract and creating a default page that documents the service and provides a facility for testing it. The Web Service can be consumed in multiple ways. It supports invocation via HTTP Get requests, HTTP Post requests, or HTTP Post requests with SOAP XML payloads. The developer requires no special knowledge of HTTP or XML.

ADO has been rewritten and is now ADO+

Now, recordsets are almost gone. The new central object is the data set. It's basically an in-memory copy of data and is similar to a recordset, but allows you to do so much more.

18.11 ASP+ Deployment

It is a very complicated process to deploy a complex ASP application that utilizes COM objects into a shared multi-application environment.ASP+ has simplified the deployment model to just copying all files which comprise an application into the appropriate directory. No registration of objects or restarting of the application is required. All parts of an ASP+ application can be deployed in this fashion, including pages, Web Services, compiled components (DLL), and even some configuration data, as shown in the next section. For compiled components there is now a special directory under each application root, called '/bin.' Any component placed in this directory is automatically made available for use within that application. Note that they are made available only to that specific application. This gives you true application isolation—that is, each application on the server could be running a slightly different version of the component with no effect on other applications! This is a boon for developers who host multiple instances of a site on one machine, such as a development environment, a test environment, and a staging environment, each with different component versions. ASP+ no longer locks DLLs to prevent you from updating them. You can just copy the new DLL on top of the old one. ASP+ automatically detects the change and migrates to the new version. Currently executing requests are drained off the old version and new requests are directed to the new version. You no longer have to reboot or restart the application. To uninstall, you simply delete any piece of the application .You need not unregister components or enter Windows Registry to change.

18.12 ASP.NET Features

In ASP.NET almost every element can be accessed at server side as runtime component. For example there is a 'page' object that is the runtime representation of ASP Page itself. Hence, ASP.NET offers separation of code from content leading to better manageable and reusable code.

18.12.1 Server Controls

ASP.NET introduces the concept of Server Controls whereby we can manipulate the values of the controls on the server side. A control, say a text box, can be made server side control by adding runat='server' attribute to it, as illiustrated below.

```
<asp:TextBox id="myTextControl" runat="server" />
```

A significant feature of the server side controls is that these controls can fire server side events. Thus the control appearing at the client, with Runat attribute set to server, can fire server side events. This is particularly useful when validating forms, or generating them dynamically. The server side controls can be subdivided into the following categories based on the functionality they offer.

Intrinsic Controls

These controls create HTML elements on the client browser with the added capability of maintaining the state

1. **List Controls**: These Controls are used to display a wide variety of lists on the client side
2. **Rich Controls**: These are used to output HTML in rich format to the client. The calendar control is one example of this category

3. **Validation Controls**: These make the art of server side validation very simple.
4. **Session State**

In ASP.NET, the Session state is automatically maintained by using a hidden field called viewstate. Further, ASP.NET offers the flexibility to turn the session state on or off at both the page level as well as at the control level. It is also possible to store the session state directly into SQL Server database and retrieve it at a later time.

Form Validation

ASP.NET offers rich set of server controls to achieve the task of Form Validation. These controls popularly known as Validators or validation controls can be used to validate all sort of validations.

```
<input type="text" id="serialNumber" runat="server" />
<asp:RequiredFieldValidator id="serialNumberVal" runat="server"
    ControlToValidate="serialNumber"
    display="static"> Error: Please provide the serial number
</asp:RequiredFieldValidator>
```

The above example of a validator control shows a required field validator control 'bound' to a text box. If the user did not enter any value in the text box, the validator would raise an error. In the above case the error message is a static text.

Browser Detection

When a web page is being requested by a client the server has to send the HTML Content depending on the client browser version. In ASP it used to be taken care programmatically by browser capabilities component but in ASP.NET it is being automatically taken care by server components present in it.

<%@ Page ClientTarget="Downlevel"%>

The above page directive indicates that any validation has to be on the server side. This, effectively, is for older browsers that may or may not support client-side scripting. The opposite of this, is:

<%@ Page ClientTarget="Uplevel"%>

Downlevel is the default, if ClientTarget is not specified.

18.13 Web Services

ASP.NET offers a new concept called Web Services by which a web application can expose functionality to others on the Internet by employing a protocol such as SOAP (Simple Object Access Protocol) and in XML format. This is entirely a new concept andhas scope for further exploitation. Web Services can be employed for display of news headlines, weather reports etc. given that the source site exposes these functionality to the outside world.

18.14 User Controls (Formerly Pagelets)

ASP.NET offers code reusability by introducing the concept of 'User Controls.' A User Control is a pseudo control, wrapped with common code, that can expose properties, methods and events.These can be employed for opening and closing database connection, closing connections

which happens quite frequently in all the pages. User Controls can be profitably employed to wrap up the most frequently used html code and reuse across several Web pages.

An extremely simple example of user control is as follows. Here we create an user control for a textbox, and use it in other pages. The file is called sample_pagelet.ascx (ASCX is the extension for user controls)

```
<script language="VB" runat=server> Public TextName as String
</script>
<input type=textbox name="<%=TextName%>" size=10>
```

This 'control' can then be used on other ASP.NET pages in the following manner:

```
<%@ Register TagPrefix="myUserControl" TagName="myText"
Src="sample_pagelet.ascx"%>
<myUserControl:myText TextName="PhoneNumber" runat="server"/>
```

User controls have counterparts in ASP world—which include files. But user controls are much more powerful and flexible.

18.15 Registration of Components

In web development it is a common practice to put the business logic in COM Components. This concept effects performance boost when business objects are employed, due to the fact they are compiled. But the maintenance aspect posed a problem. If you want to unregister a component in use by the web server, you will have to restart the server and then unregister the component.

ASP.NET has the solution for this problem. If you want to register a component in ASP.NET, all you need to do is to copy the component in web applications 'bin' directory. To unregister the component, you just delete the file.

18.16 Mobile Controls

ASP.NET is in tune with M-Commerce by providing .NET Mobile Web SDK which allows us to build applications covering wide range of mobile devices ranging from cellular phones to personal Digital assistants. The versatility of these controls lie in that they get rendered based on the hand held device employed to view them.

18.16.1 Data Access

Finally with regards to data access ASP.NET introduced the next generation of ADO known as ADO.NET. ADO.NET places more emphasis on disconnected recordsets by employing XML as a medium of communication between these recordsets and DataStore.

18.16.2 Error Handling and Debugging

ASP.NET Offers a cleaner approach to error handling and debugging as compared to its predecessor ASP. It is now possible to specify individual error pages for each ASP.NET page by employing the new ErrorPage directive.

```
<%@page Errorpage = "/myErrorPage.aspx"%>
```

3. Valid
4. Sessio

In ASP.NE
Further, A
as well as
Server dat

Form Vali
ASP.NET c
popularly k

```
<input
<asp:I
    Co
    d:
</asp
```

The above
text box. If
In the abov

Browser D
When a we
depending
browser ca
component

<%@ Page C

The above
effectively, i
of this, is:

<%@ Page C

Downlevel i

18.13 Wel

ASP.NET of
functionality
Access Protc
exploitation.
given that th

18.14 Use

ASP.NET offi
a pseudo co
events.Thes

In the above example, if any error occurs on the page, t
This type of error handling enables developers to circu

Error handling features are further enhanced by prov
statement in VB.NET. ASP.NET Pages have the Trac
the controls and variables can be sent to the client a
disabled programatically.

18.16.3 User Authentication

ASP.NET offers several kinds of authentication for chec
a web site. In addition to support for Basic, Digest an
supports Passport authentication

18.16.4 Server Side Caching

The Process of keeping most frequently visited web page
of ASP.NET Page leads to an increase in its performanc
types of caching

- Output caching
- Data Caching

Output caching is the process of caching the entire pag
sites with heavy traffic. Data caching is the process of c
objects and data.

18.17 Configuration Details

ASP.NET Maintains all the configuration details pertai
Config.web which is in a human readable format (XML).
including which request to handle on specific situation
file.

In ASP.NET its possible to specify the event handlers th
particular events. These event handlers are defined in
Global.asax file of ASP.NET has a provision for the follo

- Application_OnStart
- Application_OnEnd
- Session_OnStart
- Session_OnEnd
- Application_BeginRequest
- Security_OnAuthenticate

The features offered by ASP.NET are too many and expla
There is lot to explore in world of ASP.NET and no dou
exploring these cool and great features.
Watch .net101 for more articles on ASP.NET in days t
every feature we have talked in the discussion above.

rovides a powerful framework for developing and deploying Next Generation
ASP.NET will have a significant impact on the web application development
ie".

Started with ASP.NET

and Install the .NET SDK

ded as part of the .NET SDK, which is a collection of all the technologies
build, and test .NET Framework-based applications. To download and install
to the Download section of MSDN. The download is about 111 MB; there is,
to download the SDK as a series of smaller files.

ASP.NET, you need to install Internet Explorer 5.5.

the technology preview of the SDK supported only Windows 2000 Server
net Information Services (IIS) 5.0, the Beta 1 release supports Windows NT

.NET Framework Quickstart Samples

d the SDK, you should also install the .NET Framework Quickstart Samples.
SDK, the Quickstart samples contain detailed examples of various ASP.NET
.NET technologies.

kstart samples:

Overview link ('.NET Framework SDK Overview'), which is added to your
the SDK installation.

IL page is open, follow the 'The .NET Framework Samples' link.

llation steps to install the samples.

he installation, you can access the samples at http://localhost/quickstart/
you navigate the default document, you can select the ASP.NET link at the
s link contains a listing of the ASP.NET samples.

ASPNG Discussion List

s of a growing group of ASP.NET community leaders,hosts an e-mail-based
SP.NET; in fact, it has been available ever since the Professional Developers
was launched in early July.

is one of the best places to learn and contribute to the growing pool of
surrounding ASP.NET. Charles monitors and filters all messages to ensure
re relevant to the list, so you'll never get any spam or off-topic discussions.
mmunity aspects of the list, the Microsoft .NET Framework team regularly
discussions and posts sample code. Most recently, Scott Guthrie posted a
eration ASP.NET sample.

y Asked Questions

application break if the SDK is installed and ASP.NET used? No, ASP.NET
side-by-side with existing ASP applications.

The file extensions used by ASP (.asp, .asa) and those used by ASP.NET (.aspx, .asax, .ascx, .asmx, etc.) are completely separate and do not override each other. Furthermore, a separate ISAPI extension handles the processing of ASP.NET applications.

One caveat: Although ASP and ASP.NET applications can run side-by-side, within the same directory, they share no application information. For example, settings made within global.asa will not apply to ASP.NET applications; settings made in global.asax will not apply to ASP applications. Similarly, Application and Session state information is not shared. Session data added for a user in an ASP.NET application is not available for that user in an ASP application unless it is re-created.

Is Visual Studio or some other tool required to create ASP.NET pages?

No. Although Visual Studio.NET will provide an amazing design, coding, and debugging environment, it is not required to build ASP.NET applications. ASP.NET follows the same 'just hit save' design principle of ASP: Any text-based editing application may be used to create ASP.NET applications. When an ASP.NET page is first accessed, it is compiled automatically, and future requests will access the compiled resource.

Does ASP.NET run on Windows 95, Windows 98 and Windows Me? No. You can certainly use one of these platforms as your development machine, but your ASP.NET application must run on a Web server, either Windows 2000 or Windows NT 4.0, with IIS installed.

Step 5 - In this step an outline of some of the great resources already available online, in print, and in person will be found.

IBuySpy Microsoft ASP.NET Sample Application IBuySpy is a full-feature, e-commerce application that provides an excellent introduction to building high-performance, scalable Web applications by using ASP.NET. You can browse the sample application online, including full source code and documentation, or you can download a complete C# or Visual Basic.NET version of the application.

ASP.NET Hosting Brinkster Inc., the largest free ASP Web host, is now offering free ASP.NET hosting. Members get 5 MB of space to test and learn ASP.NET. The membership is for testing and learning purposes, and will give developers an opportunity to explore this new technology. Brinkster has been offering free ASP hosting for more than a year and is a leader in the ASP/ ASP.NET hosting market.

Community Sites and Distribution Lists the community (people, events, discussion lists, newsgroups, etc.) forming for ASP.NET is one of the best places to get questions answered and problems solved. A few of the sites are hosting Quickstart samples online—where you can try ASP.NET, but never install it.

http://www.aspfree.com/_This is one of the more comprehensive ASP.NET community sites, run by Steve Schofield. Here you can find a dozen articles on ASP.NET, along with the ASP.NET Quickstart samples online; a great alternative for learning ASP.NET if you can't install the .NET SDK. As a bonus, chapter 1 from the Wrox Press book, A Preview of ASP.NET, is available online as well.

http://www.aspng.com/_ Is another comprehensive ASP.NET related sites. ASPNG includes links to ASP.NET classes hosted by Charles Carroll and ASP.NET workshops hosted by Stephen Walther. Finally, ASPNG hosts one of the more active e-mail discussion lists (see http:// www.asplists.com/).

18.19 Requirement for running ASP.NET

To run ASP.NET, you must have the .NET Framework SDK. This beta software should only be installed on non-production test machines. The SDK will run on Windows(r) 2000 and Windows NT(r) 4.0 (including the NTOP for ASP.NET support), Windows 98, or Windows ME. Internet Explorer 5.5 and Data Access Components 2.6 are required.

The command line compilers for VB, C# and JScript are available with ASP.NET in addition to VS.Net. This means that you don't have to purchase/install VS.Net in order to write components with ASP.NET. You can simply install the "SDK" version of ASP.NET that will be available publicly on the Microsoft.com website and go from there. ASP.NET and the .NET Frameworks can be downloaded, installed and used on both development and deployment servers for free.

ASP.NET will provide multiple types of HTML depending on the browser. 'UpLevel' browsers (IE4.0+) will get HTML4.0 and Client- Side JavaScript 1.0. All other browsers ('DownLevel') will get HTML3.2 and no JavaScript. That is as of Beta 1

Once released, you will be able to run ASP.NET on either IIS5 or IIS4 (we will not require a new version of IIS to install it). Because we run side-by-side with ASP, there shouldn't be any real adoption blockers to installing it.

You can use VS6 to edit ASP.NET Applications.You may also use notepad or VS7 (which has WYSIWYG Designers and Debuggers for VB, C# and C++ that target ASP.NET)

18.20 How is ASP.NET different from ASP 3.0?

ASP.NET is a completely new ASP. It was written entirely from the ground up (homage to Scott Guthrie and Mark Anders for their ingenious idea)in C# (pronouuced 'see sharp'). ASP.NET makes use of compilable programming languages, such as Visual Basic.Net, C#, even COBOL! (making it truely language-neutral), to write Web Applications. The applications are compiled on the server, and pages are served in HTML for the specific browser that has issued the request

18.21 Create Dynamic Content Using ASP

Microsoft's Active Server Pages (ASP) technology is a powerful tool for building Web sites that incorporate dynamic content, like pages built from databases. It will be particularly beneficial to developers familiar with Microsoft's Visual Basic. Microsoft presents ASP as the server half of an overall Active Platform; the other half—Active Desktop— represents the client side. Both server and desktop use similar technologies:

- Tools.
- HTML.
- VBScript and JScript (Microsoft's version of JavaScript).
- Components (ActiveX and Java).
- System services (like component and directory services on the server and audio, video, and file system on the desktop).

In implementation, however, the active server is entirely independent of Active Desktop—it is a set of technologies that runs on an NT server running Internet Information Server (IIS). The Active Platform appears to be more of a "vision thing" or marketing concept than a development concern. For example, you do not have to use ActiveX controls in the pages that ASP serves up.

18.22 Working with Properties and Data Types in ASP.NET

In classic ASP a color name and a string didn't seem any different in that they were both string values. In .NET data types are much more important and much more strict.

Every class (data type) in .NET exposes a set of properties and methods. The solution to the problem is found in these properties and methods. In this case the goal is to convert a String data type to a Color data type. The System.Drawing.Color class provides this functionality with its FromName method. The FromName method takes in a string value as its only parameter. The value passed in is then used to create a new color object. Listing 1.1 shows an ASP.NET Web Form that uses a DropDownList of string values to trigger a PostBack and alter the ForeColor property of a Label control.

Source Code

Altering the ForeColor Property of a Label Control

Listing 1.1

```
<% @ Page Language="VB" %>< /FONT>
<head><script runat="Server">< /FONT> Sub Page_Load(Source As Object,
E As EventArgs)
If     Page.IsPostBack    Then    Message.ForeColor    =
System.Drawing.Color.FromName(ColorPick.SelectedItem.Text) End If End
Sub
</script></head>
<html>
<body>
<form runat="server" method= "post">< /FONT> Pick a color: asp:
<asp:DropDownList runat="server" id="ColorPick"
AutoPostBack="True" >< /FONT>
<asp:ListItem Text="Black" />< /FONT>
<asp:ListItem Text="Red" />< /FONT>
<asp:ListItem Text="Blue" />< /FONT>
< asp:ListItem Text="Green" />< /FONT>
<asp:DropDownList />
<asp:Label runat="server" id="Message" Text="This is the result
of the colorpicker" />< /FONT>
</form>
</body>
</html>
```

The Page_Loadevent handler is used to catch and handle the PostBack event with If Page.IsPostBack Then... In the event of a PostBack you make a call to the FromName method of the System.Drawing.Color class. You pass in a single parameter, a string value. The value passed in is the SelectedItem value of the DropDownList. The FromName() method takes the value in and instantiates a new Color object based on the value.

18.23 Facilites Provided by ASP.NET

18.23.1 WebForm Framework

Since the Web Forms framework creates an abstraction of the traditional browser-Web server interaction, you can program your application using traditional RAD and OOP: you can work with HTML elements using properties, methods, and events. The framework overcomes the separation of client and server inherent in Web- based applications, presenting a unified model for responding in server code to client events, and for automatically maintaining the state of a form during forms processing. Further, Web Forms supports a server control framework that encapsulates common UI functionality in easy-to-use (and reusable) controls.

18.23.2 Web Services

A Web service is a piece of business functionality exposed through Internet protocols. It enables applications to exchange information between Web-based applications using standards like HTTP and XML messaging to move data across firewalls. Web services are not tied to a particular component technology or object-calling convention. As a result, programs written in any language, using any component model, and running on any operating system can access Web services.

18.23.3 State Facilities

ASP.NET provides built-in session state functionality. In addition, you can manage application-specific values using explicit state management facilities. This is useful for storing information that is independent of controls or that must be shared between pages.

ASP.NET application state values are only accessible from code running within the originating application's context. Other applications operating on the same system cannot gain access to or modify them.

ASP.NET also offers distributed state facilities. You can create multiple instances of the same application on one computer or on several computers.

18.23.4 Application Events

ASP.NET allows you to include application-level event handling code in the optional global.asax file.These may be used to manage cross-session information and perform orderly application startup and cleanup tasks. For more details, see Web Forms Event Model.

18.23.5 Compilation

All ASP.NET code, including scripts, is compiled, which helps strong typing, performance optimizations, and early binding, among other benefits. Once the code has been compiled, the common language runtime further compiles ASP.NET to native code, providing dramatically improved performance. For more details, see Web Forms Compilation, Deployment, and Execution.

18.23.6 Deployment

ASP.NET provides a simple mechanism for deploying applications to local or remote servers. You can install files by simply copying them to the appropriate application directories. Although ASP.NET pages, services, and components can be compiled, the compiled files do not need to be registered on the installation server.

ASP.NET configuration settings are stored in XML-based files, which are human readable and writable. Each of your applications can have a distinct configuration file and you can extend the configuration scheme to suit your requirements.

In Visual Studio, the project framework handles deployment of your Web Forms projects for you, including copying them to remote servers. For more details, see Web Forms Compilation, Deployment, and Execution.

18.23.7 Security

ASP.NET provides default authorization and authentication schemes for Web applications. You can easily remove, add to, or replace these schemes depending upon the needs of your application.

18.23.8 Manageability

ASP.NET takes advantage of the common language runtime debugging infrastructure to provide cross-language debugging support that can be used both locally and remotely from a Web server. In addition, the Web Forms page framework provides a trace mode for inserting instrumentation messages into your forms. For details, see Web Forms Testing and Debugging.

ASP.NET also supplies performance counters within the Windows Performance Monitor itself. These counters can be used to monitor the performance of a single instance of an ASP.NET application.

18.24 Introduction to Web Forms

Web Forms are an ASP.NET technology that can be used to create programmable Web pages. They use HTML to present information to the user in any browser and code on the server to implement application logic.

Web Forms use a programming model that can be used on the server to dynamically generate Web pages. Web Forms so created have the following features:

1. They can run on any browser and automatically render the correct, browser-compliant HTML for features such as styles, layout, and so on. Alternatively, you can design your

Web Form pages to run on a specific browser such as Microsoftr Internet Explorer 5 and take advantage of the features of a rich browser client.

2. They can be programmed in any .NET framework language, including Microsoftr Visual Basic, Microsoftr C#, Microsoftr Managed Extensions for C++, and Microsoftr JScript.NET.
3. They are built on the .NET framework and provide all the benefits of those technologies, including a managed execution environment, type safety, and inheritance.
4. They are supported in Visual Studior with powerful RAD developmen tools for designing and programming forms.
5. They support a rich set of server-based controls that provide standard forms capability for Web development in a rapid application development designer. and are extensible with user-created and third-party controls.

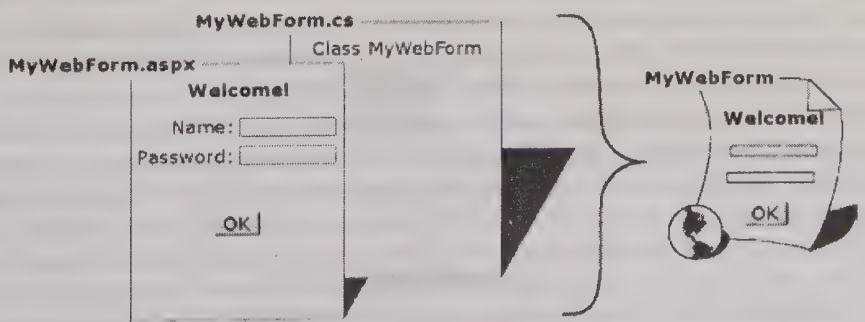
18.24.1 Components of Web Forms

Web Forms divide the Web applications user interface into two pieces: the visual component and the user interface logic. If you have worked with tools like Visual Basic and Visual C in the past, you will recognize this distinction between the visible portion of a form and the code that interacts with the form.

The user interface for Web Forms pages consists of a file containing HTML and Web-Forms-specific elements. This file is referred to as the page. The page works as a container for the text and controls you want to display. Using the Visual Studio HTML Editor plus Web Forms Server Controls, you can lay out the form as you might in any Visual Basic application. The page is a file with the extension '.aspx.'

The user interface logic for the Web form consists of codes that can be created to interact with the form. The programming logic resides in a separate file (referred to as the 'code-behind' file), written in Visual Basic or C#. When you run the form, the code-behind class file runs and dynamically produces the HTML output for the page.

Web forms consist of a page for visual components and a code-behind class file



Both files together make up MyWebForm.

18.24.2 Application of Web Forms

Web application programming presents challenges that do not typically arise when programming traditional client-based applications. Among the challenges are these:

Rich user interface: A user interface with a large amount of content, a complex layout, and rich user interaction can be difficult and tedious to create and program using basic HTML facilities. It is especially hard to create a rich user interface for applications likely to run in many different browsers.

Separation of client and server: In a Web application, the client (browser) and server are different programs often running on different computers (and even on different operating systems). Consequently, the two halves of the application share very little information; they can communicate, but typically exchange only small chunks of simple information.

Stateless execution: When a Web server receives a request for a page, it finds the page, processes it, sends it to the browser, and then, effectively, discards all page information. If the user requests the same page again, the server repeats the entire sequence, reprocessing the page from scratch. Put another way, servers have no memory of pages that they have processed. Therefore, if an application needs to maintain information about a page, this becomes a problem that has to be solved in application code.

Unknown client capabilities: In many cases, Web applications are accessible to many users using different browsers. Each of these browsers has different capabilities, making it difficult to create an application that will run equally well on all of them. Data access. Reading from and writing to a data source in traditional Web applications can be complicated and resource-intensive.

Meeting these challenges for Web applications can require substantial time and effort. Web Forms address these challenges in the following ways:

1. Bowser-independent applications. Web Forms provide a framework for creating all application logic on the server, eliminating the need to explicitly code for differences in browsers. However, it still allows you to automatically take advantage of browser-specific features to provide improved performance and a richer client experience.

Event-based programming model. Web Forms bring to Web applications the familiar model of writing event handlers for events that occur in either the client or server. The framework abstracts this model in such a way that the underlying mechanism of capturing an event on the client, transmitting it to the server, and calling the appropriate method is all automatic and invisible. The result is a clear, easily-written code structure.

Abstract, intuitive, consistent object model. The Web Forms framework presents an object model that allows you to think of your forms as a unit, not as separate client and server pieces. In the Web Forms model, you can program the form in a much more intuitive way than in traditional Web applications, including the ability to set properties for form elements and respond to events. In addition, Web Forms server controls are an abstraction from the physical contents of an HTML page and from the direct interaction between browser and server. In general, you can use Web Forms server controls the way you might work with controls in a client application and not have to think about how to create the HTML to present and process the controls and their contents.

State management. The Web Forms framework automatically handles the task of maintaining the state of your form and its controls, and provides you with explicit ways to maintain the state of application-specific information. This is accomplished without heavy use of server resources and without sending cookies to the browser, two traditional means for storing state.

Scalable server performance. The Web Forms framework allows you to scale your application from one computer with a single processor to a multi-computer 'Web farm' cleanly and without complicated changes to the application's logic.

18.25 Web Services Overview

In recent years, one force has irrevocably restyled the application development landscape more than any other the Internet. Organizations increasingly depend upon the digital resources and channels of communication that are provided by the Internet and intranets. As a result, very few applications are designed and developed today that do not consider how best to incorporate and leverage Web technologies to fully leverage the benefits of connected computing.

When not pursuing wholly new application development, organizations attempt to create applications that aggregate several traditional, task-oriented applications into a single, composite application. This sometimes includes integrating applications that exist within the boundaries of a separate entity, such as another company or a service provider. However, a still greater dilemma arises when attempting to integrate legacy applications that were built using an amalgamation of technologies, object models, operating systems and programming languages. To make them all work together, the programmable Web can be used.

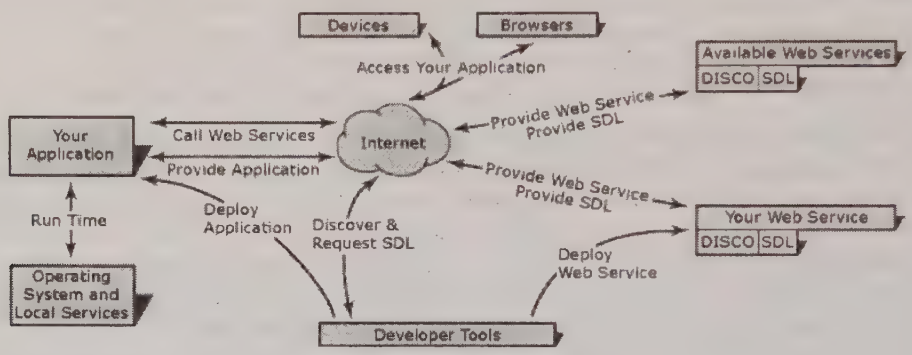
The XML (Extensible Markup Language) data description format has given rise to the reality of a programmable Web. Just as TCP/IP provided universal connectivity for the Internet and HTML provided a standardized language to display information on a wide variety of platforms for human consumption, XML provides a standardized language to exchange data for automated consumption. Representing data in a widely accepted format enables computers to send and receive data in predictable styles, which enable programmability that extends beyond closed, controlled systems. XMLendows freedom because its simplicity and extensibility let you define just about anything with scope for expansion. One of the fundamental building blocks of the programmable Web is Web Services.

A Web Service can be used either internally by a single application or exposed externally over the Internet for use by any number of applications. Web Services provide a viable solution for enabling code interoperability. Most Web Services utilize XML-based messaging as a fundamental means of data communication to help bridge the differences that exist between systems that use incongruent component models, operating systems, and programming languages. Developers can create applications that weave together Web Services from a variety of sources in much

the same way that developers traditionally use components when creating a distributed application.

Web Services are similar to components in that they encapsulate more complex functionality. However, Web Services differ from components in that they do not mandate a particular object model to access them. One of the core characteristics of a Web Service is the high degree of abstraction that exists between the implementation and consumption of a service. By using XML-based messaging as the mechanism by which the service is created and accessed, both the Web Service client and the Web Service provider are freed from needing any knowledge of each other beyond inputs, outputs, and location.

Web Services Application Model



18.26 Introduction to Web Forms Server Controls

When you create Web Forms, you can use these types of controls:

1. **HTML server controls**: HTML elements exposed to the server so you can program them. HTML server controls expose an object model that maps very closely to the HTML elements that they render.
2. **ASP.NET server controls**: Controls with more built-in features than HTML server controls. ASP.NET server controls include not only form-type controls such as buttons and text boxes, but also special-purpose controls such as a calendar. ASP.NET server controls are more abstract than HTML server controls, in that their object model does not necessarily reflect HTML syntax.
3. **Validation controls**: Controls that incorporate logic to allow you to test a user's input. You attach a validation control to an input control to test what the user enters for that input control. Validation controls are provided to allow you to check for a required field, to test against a specific value or pattern of characters, between ranges, and so on.

4. **User Controls**: Controls that you create as Web Forms pages. You can embed Web Forms User Controls in other Web Forms pages, which is an easy way to create menus, toolbars, and other reusable elements. You can use all types of controls on the same form. The following

sections provide more detail about server controls. For more information about validation controls, see Web Forms Validation.

18.27 HTML Server Controls

HTML server controls are HTML elements containing attributes that make them visible to — and programmable on—the server. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program on the server.

The object model for HTML server controls maps closely to that of the corresponding elements. For example, HTML attributes are exposed in HTML server controls as properties.

Any HTML element on a page can be converted to an HTML server control. Conversion is a simple process involving just a few attributes. As a minimum, an HTML element is converted to a control by the addition of the attribute RUNAT=SERVER. This alerts the page framework during parsing that it should create an instance of the control to use during server-side page processing.

If you want to reference the control as a member within your code, you should also assign an ID attribute to the control.

The Web Forms framework provides pre-defined HTML server controls for the HTML elements most commonly used dynamically on a page: forms, the HTML <INPUT> elements (text box, check box, Submit button, and so on), list box (SELECT), table, image, and so on. These pre-defined HTML server controls share the basic properties of the generic control, and in addition, each control typically provides its own set of properties and its own event.

18.28 Features of HTML server

It is an object model that you can program against on the server using the familiar object-oriented techniques. Each server control exposes properties that allow you to manipulate the control's HTML attributes programmatically. A set of events for which you can write event handlers in much the same way you would in a client-based form, except that the event is handled in server code. The ability to handle events in client script. Automatic management of the values of the form's controls. If the form makes a round trip to the server, the values that the user entered into HTML server controls are automatically maintained when the page is sent back to the browser. Interaction with validation controls help the user easily verify whether the appropriate information has been entered into a control.It allows Data binding to one or more properties of the control.It renders support for HTML 4.0 styles if the Web Forms page is displayed in a browser that supports cascading style sheets.It offers Pass-through of custom attributes. You can add any attributes you need to an HTML server control and the Web Forms framework will read them and render them without any change in functionality. This allows you to add browser-specific attributes to your controls.

18.29 ASP.NET Server Controls

ASP.NET server controls are a second set of controls designed with a different emphasis. They do not map one-to-one to HTML server controls. Instead, they are defined as abstract controls in which the actual HTML rendered by the control can be quite different from the model that you program against. For example, a RadioButtonList ASP.NET server control might be rendered in a table or as inline text with other HTML.

ASP.NET server controls include traditional form controls such as buttons and text boxes as well as complex controls such as tables. They also include controls that provide commonly-used form functionality such as displaying data in a grid, choosing dates, and so on.

ASP.NET server controls offer all of the features described above for HTML server controls (except one-to-one mapping to HTML elements) and these additional features:

1. A rich object model that provides type-safe programming capabilities. *Automatic browser detection. The controls can detect browser capabilities and create appropriate output for both basic and rich (HTML 4.0) browsers.
2. For some controls, the ability to define your own look for the control using templates.
3. For some controls, the ability to specify whether a control's event causes immediate posting to the server or is instead cached and raised when the form is submitted.
4. Ability to pass events from a nested control (such as a button in a table) to the container control.
5. At design time in HTML view, the controls appear in your page in a format such as:

```
<asp:button attributes />
```

The attributes in this case are not those of HTML elements. Instead, they are properties of the ASP.NET control.

When the Web Forms page runs, the ASP.NET control is rendered on the page using appropriate HTML, which often depends not only on the browser type but also on settings that you have made for the control. For example, a Button control might render as an <INPUT> tag or a <BUTTON> tag, depending on its properties.

18.30 Adding HTML Server Controls to a Web Forms Page

Web Forms Server Controls | Adding Controls to a Web Forms Page Programmatically | Adding ASP.NET Server Controls to a Web Forms Page | Converting HTML Server Controls to HTML Elements | Creating Web Forms Event Handling Methods in the Designer | Setting HTML Server Control Properties Programmatically | Web Forms Validation

[This topic is part of a beta release and is subject to change in future releases. Blank topics are included as placeholders.]

Adding HTML server controls is like adding any HTML element, except that you convert the element to a control so that you can work with it in server code.

Note: For background information about controls for Web Forms pages, see Introduction to Web Forms Server Controls. Adding HTML Using the Visual Studio Web Forms Designer Adding HTML server controls to a Web Forms page is a two-stage process.

To add an HTML server control to a Web Forms page

1. From the HTML tab of the Toolbox, drag an HTML element onto the page.
2. Convert the element to a control by right-clicking it and choosing Run As ServerControl.
3. The Web Forms Designer adds the attribute RUNAT=SERVER to the element, which alerts the server to treat the element as a control. A glyph appears on the control in Design view to indicate that it is a server-based control.

Note: By default, the Web Forms page uses HTML linear layout. If you want to place controls at absolute positions on the page —using x and y coordinates—you need to place them using a Panel (Grid Layout) HTML server control.

18.31 Adding HTML Server Controls using ASP.NET Syntax

You can add an HTML server control to a page by declaring it in directly in the .aspx file.

To add an HTML server control using ASP.NET syntax

Type the HTML syntax of the element you want to use as a control. Include the normal HTML syntax for the element, and in addition, do the following:

1. Set the control's ID attribute (property) to a unique value for tha page, unless the control is part of a complex control and will be repeated (as in Repeater, DataList, and DataGrid controls).
2. Set the runat=server attribute to convert the element to a control.

Note: In Visual Studio, you do this in HTML view. The following example shows how to declare an HtmlInputText control, an HtmlInputButtoncontrol, and an HtmlAnchor control:

```
<input id="Name" type=text size=40 runat="server">
<input type=submit value='Enter" runat="server"> Click <a id="Anchor1"
runat="server">More </a> to see the next page.
HtmlInputText Control  [C#]
```

The HtmlInputText control allows you to run server code against the HTML <input type=text> and <input type=password> elements. As with standard HTML, these controls can be used to enter user names and passwords in HTML forms.
You can use this control in conjunction with the HtmlInputButton, HtmlInputImage, or HtmlButton control to process user input on the server. You can retrieve or assign values to the HtmlInputText MaxLength, Size, and Value properties.

Note This control does not require a closing tag.

Tasks

Verifying User-Entered Contents of a Text Box Against Stored Values

ASP.NET Syntax

Required properties are noted in boldface type.

```
<input
   type=text | password
   runat="server"
```

```
    id="programmaticID"
    maxlength="max#ofcharacters"
    size="widthoftextbox"
    value="defaulttextboxcontents"
>
Verifying User-Entered Contents of a Text Box Against Stored  Values
```

The following example shows a simple authentication scenario. This example uses two values from the text boxes on the Web Forms page. A more advanced scenario would be needed to verify the entered text against values stored in a database.Text boxes in combination with validation controls helps verify that the correct information is being entered in your text boxes.

HtmlInputButton Control [C#]

The HtmlInputButton control allows you to program against the HTML <input type=button>, <input type=submit>, and <input type=reset> elements. When a user clicks an HTML Input Button control, input from the form where the control is embedded is posted to the server, processed there, and a response is sent back to the requesting browser.

Used in conjunction with the HtmlInputText and HtmlTextArea controls, you can create user input or authentication pages that can be processed on the server.

> **Note: This control does not require a closing tag.**

ASP.NET Syntax Required properties are noted in boldface type.

```
<input
    type=button | submit | reset
    runat="server"
    id="programmaticID"
    OnServerClick="onserverclickhandler"
    >
```

Responding to HtmlInputButton Control Events

The following code provides a number of useful examples of simple event handling. The file processes a simple authentication scheme by comparing text entered by the user with text contained in event handling code in the head of the HTML document. The body of the HTML page contains an HtmlForm control, which in turn contains two HtmlInputText controls, two HtmlInputButton controls, and an HtmlGenericControl. The first input text control is a simple text box, while the second is a password box; the two buttons are submit and reset, respectively. The generic control is the element that displays the appropriate message specified by the event handling code.

The SubmitBtn_Click code uses a conditional loop to respond in two possible ways when the user clicks the Submit button. First, if the user enters JohnDoe in the text box and GoldFish in the password box and clicks the submit button, the server sends the response message of You are authenticated! to the span control. If the user enters a name or password different from these, the page returns Incorrect user name of password. to the span control.

The ResetBtn_Click code simply clears the both text boxes in the form.

HtmlAnchor Control [C#]

The HtmlAnchor control allows you to program against the HTML <a> element. This control must have an opening and closing tag. You can use this control to dynamically modify the attributes and properties of the <a> element, use a data source to generate these controls, and use control events to generate the HtmlAnchor control dynamically.

Target values must begin with a letter in the range A-Z or a-z, except the following special values that begin with an underscore: _blank, _self, _parent, and _top.

Tasks Generating the HRef Attribute Dynamically:

Data Binding to the HtmlAnchor Control

ASP.NET Syntax Required properties are noted in boldface type.

```
<a
   runat="server"
   id="programmaticID"
   href="linkurl"
   name="bookmarkname"
   OnServerClick="onserverclickhandler"
   target="linkedcontentframeorwindow"
   title="titledisplayedbybrowser"
> linktext
</a>
```

Generating the HRef Attribute Dynamically

You can dynamically generate the URL to which you want the HtmlAnchor control to link.

To generate the HRef property dynamically

Declare an HtmlAnchor control in an HTML document. For example:

```
<a id="anchor1" runat="server">.
```

Note:Remember to embed the HtmlAnchor control inside the opening and closing tags of an HtmlForm control.

Write event handling code that assigns a URL to the HtmlControl HRef property. For example, the following code generates the URL www.microsoft.com when the Page_Load event occurs.

```
Void Page_Load(Object sender, EventArgs e) {
    anchor1.HRef =;
}
```

data Binding to the HtmlAnchor Control7

You can also generate hyperlinks on a Web Forms page by binding the HtmlAnchor control to a data source.

The following example shows how to include an HtmlAnchor control within a repeating control, then data binding to generate the URL to link to, as well as the anchor text to be displayed between the opening and closing tags of the <a> element.

```
[C#]
<%@ Import Namespace="System.Data" %>
<html>
<script language="C#" runat="server"> void Page_Load(Object Src, EventArgs
E) {
        DataTable dt = new DataTable();
        DataRow dr;
       dt.Columns.Add(new DataColumn("IntegerValue", typeof(Int32)));
       dt.Columns.Add(new DataColumn("StringValue", typeof(string)));
            dt.Columns.Add(new DataColumn("DateTimeValue",
typeof(DateTime)));
        dt.Columns.Add(new DataColumn("BoolValue", typeof(bool)));
      dt.Columns.Add(new DataColumn("CurrencyValue",typeof(double)));
        for (int i = 0; i < 9; i++) {
            dr = dt.NewRow();
            dr[0] = i;
            dr[1] = "Item " + Int32.ToString(i);
            dr[2] = DateTime.Now;
            dr[3] = (i % 2 != 0) ? true : false;
            dr[4] = 1.23 * (i+1);
            dt.Rows.Add(dr);
        }
        MyRepeater.DataSource=new DataView(dt);
        MyRepeater.DataBind();
    }
</script>
<body> <h3><font face="Verdana">DataBinding HtmlAnchor</font></h3>
    <p>
    <form runat=server>
    <font face="Verdana" size="-1">
        <asp:Repeater id="MyRepeater" runat="server">
            <template name="ItemTemplate">
                Link for
                <a href='<%# DataBinder.Eval(Container,
               "DataItem.StringValue","detailspage.aspx?id={0}")  %>'
runat="server">
           <%# DataBinder.Eval(Container, "DataItem.StringValue") %>
                </a>
                <p>
            </template>
        </asp:Repeater>
    </font>
    </form>
</body>
</html>
```

18.32 Positioning HTML Elements in Design View

In the Design View of the HTML Designer, you can position elements on your HTML page in, using either LinearLayout or GridLayout.

The Page Layout property determines element positioning preferences. You can set this property from either the Document Property Page or the Properties window.

18.32.1 Linear Layout

When the pageLayout property is set to LinearLayout, the client Web browser positions elements one after the other, as they occur in the markup for the page. Elements are displayed inline with the HTML text around them. This means that elements flow from left to right within a line, and from bottom to top within the page. Elements are sometimes repositioned when the page is resized.

To position elements within the linear flow of the page, you can use the block positioning tags <CENTER>, <LEFT>, and <RIGHT>, or place elements inside a <TABLE> or <DIV>. However, you must work within the limitations imposed by these older formatting options. For example, you cannot make elements overlap, or use absolutely element positioning.

Note: Any Web browser can display HTML pages that use Linear Layout positioning.

18.32.2 Grid Layout

When the pageLayout property is set to GridLayout, elements are displayed at specified locations on the page. When you drag an element onto the page, and if the 'If Snap-to-Grid' is turned on, the element is positioned at the X opr Y co-ordinates of the drop point.

Note Setting the PageLayout property to GridLayout in the Property Page for an HTML document permits absolute positioning of its elements. An MS POSITIONING='GridLayout' attribute is added to the opening <BODY> tag, and the positioning grid is activated in Design View.

18.32.3 Snap-to-Grid

To access the Snap to Grid options,first open the Options dialog box from the Tools menu, and then select the Display page under HTML Designer.

When you turn on 'Snap to Grid', the upper left corner of a new element inserted or dropped on your Web page is aligned automatically with the positioning grid lines nearest to the insertion point.

You can also set the Horizontal Spacing and Vertical Spacing increments between grid lines. These settings adjust how Snap to Grid affects resizing behavior. When you click and drag an element's positioning handles to resize it, the element is resized in grid increments. So, for example, if both the grid spacing measurements are set to 4 pixels, an element will be resized in 4 pixel increments.

When an HTML element is placed within a containing element while 'Snap-to-Grid' is turned off, and then 'Snap-to-Grid' is turned on, the contained element is not automatically repositioned. To align the contained element with the positioning grid of its container, select Align from the

Format menu and choose an alignment option. If you need both to align and to resize a contained element, align the element first and then resize it.

18.32.4 Layering

Absolutely positioned elements are displayed at their assigned positions whether or not other elements share the same space. This means you can layer elements.

If the targetSchema property is set to HTML 3.2, overlap indicators () appear in Design View, placed at the upper-left corner of elements to warn you that elements cannot overlap in earlier versions of HTML. On an HTML 3.2 page, one or more TABLE elements are genereated to position elements. On an HTML 4.0 page, positioning is done using the CSS positioning mechanism.

18.32.5 Combining Grid and Linear Layout

In some cases, you might want to combine page layout types within the same page. For example, container objects such as a <TABLE> or a <DIV> can be absolutely positioned on the page. The objects within such containers are then positioned relative to the container's location. You can also drag both Panel (Linear Layout) and Panel (Grid Layout) elements onto a page from the Toolbox, and then place other elements inside each type of Panel.

Each Panel that you drag onto a page displays sizing handles, so that you can resize it on the Design View editing surface.

18.33 Adding Controls to a Web Forms Page Programmatically

Often it is more practical to create a control at run time than at design time. For example, imagine a search results page in which you want to display results in a table. Because you don't know how many items will be returned, you want to dynamically generate one table row for each returned item.

> **Note: Existing controls can often provide the functionality you get from creating controls dynamically. For example, controls such as the Repeater, DataList, and RadioButtonList controls can dynamically create rows or other control elements when the page runs. In order to programmatically add a control to a page, there must be a container for the new control. For example, if you are creating table rows, the container is the table. If there is no obvious control to act as container, you can use a Panel ASP.NET server control.**

In some instances, you might want to create both static text and controls. To create static text, you can use either the LiteralControl object or a Label ASP.NET server control. You can then add these controls to the container as you would any other control.

To add a control to a Web Forms page programmatically, create an instance of the control and set its properties:

```
Label myLabel = new Label(); myLabel.Text = "Sample Label"
```

Tip: Controls are typically added to the page during the page's load stage.

Add the new control to the Controls collection of a container already on the page:

```
Panel1.Controls.Add(myLabel);
```

Note: Controls that you add dynamically to a Web Forms page do not automatically become part of the page's view state—neither the controls nor their values are saved when a page performs a round trip to the server. You are therefore responsible for saving the state of any dyanamically-generated controls whose values you want to preserve.

The following example shows a complete Page_Load method that creates as many label controls as the user has selected from a drop-down list called DropDown1. The labels display the word 'Label' plus a counter. The container for the controls is a Panel ASP.NET control called 'Panel1.'

```
//C# protected void Page_Load(Object Src, EventArgs E ) {
    // Get the number of labels to create
    int numlabels = int.FromString(DropDown1.SelectedItem.Value);
    for (int i=1; i<=numlabels; i++)
    {
        Label myLabel = new Label();
        // Set the label's Text and ID properties
        myLabel.Text = "Label" + Int32.ToString(i);
        myLabel.ID = "Label" + Int32.ToString(i);
        Panel1.Controls.Add(myLabel);
        // Add a spacer in the form of an HTML <BR> element
        Panel1.Controls.Add(new LiteralControl("<br>"));
    }
}
```

18.34 Web Forms Page Processing

The life cycle for a Web Forms page is, generally speaking, similar to that of any Web process that runs on the server. Certain characteristics of Web processing—information passed via HTTP protocol, the stateless nature of Web pages, and so on—apply to Web Forms just as they do to other Web applications.

However, the Web Forms framework performs many Web application services for you. As one example, the Web Forms framework captures the information posted with the form, extracts the relevant values, and makes it available to you in object properties.Even so, it is helpful to understand the sequence of events that occurs when a Web Forms page is processed for programming Web Form pages effectively.

18.35 The Life Cycle of a Web Forms Page

Before you examine the details of what goes on inside a Web Forms page when it is processed, you might find it helpful to understand some fundamental characteristics of how Web Forms work as Web applications.

Round Trips

One of the most important things to understand is the division of labor in a Web Forms page. The browser presents the user with a form, and the user interacts with the form. However, all processing that interacts with other server components must occur on the server. This means that for each action that requires processing, the form must be sent (posted) to the server, processed, and returned to the browser. This sequence is referred to as a round trip.

Note: You can create client script in Web Forms, which is useful for user input validation and for some types of UI programming. However, client script does not interact with server components. For example, if the user enters an order and you want to confirm sufficient inventory for the order, your application posts the page to the server at an appropriate point in the user's order entry. A server process examines the order, performs an inventory lookup, perhaps takes some action (such as modifying the page to indicate an error), and then returns the page to the browser for the user to continue.

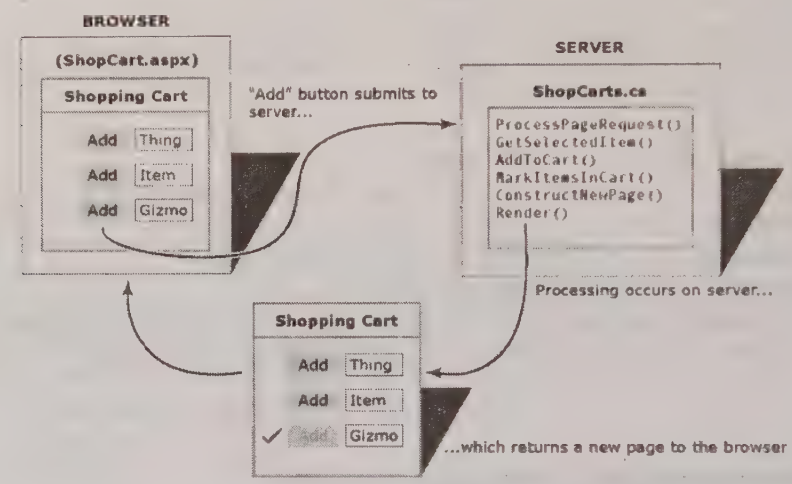


Figure label

In Web Forms, most user actions such as clicking a button result in a round trip. For that reason, the events available in Web Forms server controls are limited. Most server controls expose a click event—for example an event that requires an explicit user gesture. By the same token, server controls do not expose high-frequency events such as onmouseover, because each time the event is raised, it would require another round trip to the server, which would considerably affect response time in the form.

18.35.1 Recreating the Page (View State and State Management)

In any Web processing scenario, pages are created afresh with every round trip. In effect, as soon as the server finishes processing a page and sends it to the browser, it discards the page information. The next time the page is posted, the server starts all over in creating and processing it. For this reason, Web pages are said to be stateless; that is, the values of a page's variables and controls are not preserved on the server.

In fact, the server might cache page information to optimize the pages, but for purposes of application programming, it is clearest to think of them as being disposed of as soon as the server has finished processing them. In a traditional Web application, therefore, the only information that the server has about a form is the information that the user has filled into controls, because that information is sent to the server when the form is posted. Other information, such as variable values or property settings, have been discarded.

The Web Forms framework works around these limitations in several ways. First, the Web Forms framework saves page and control properties between round trips, which is referred to as saving the controls' view state. (The framework can detect when a form is requested for the first time versus when the form is posted, which allows you to program accordingly). In addition, the Web Forms page framework provides state management facilities so you can store other application-specific information (not just the values of controls) between round trips or to pass between pages.

18.35.2 Linear Processing

In client-side forms, the form is initialized and displayed along with its controls. Users then interact with the form, which causes events to be raised which in turn call event handlers.

In traditional server-based Web forms, because user actions cause the form to be posted to the server, and because the server must recreate the page with each round trip, processing is more linear than in a client-side form. Effectively, after being recreated, the page is processed in the same top-to-bottom sequence each time. Thus,the form is not truly event-driven.

The Web Forms framework overcomes this strict linear processing in various ways. For example, the framework allows you to set up event handlers in server code for events that occur in the browser. When the user clicks a Button ASP.NET server control, for example, the button raises an event that is transmitted via a form post to the server. There, the Web Forms framework interprets the post information and as part of its normal processing calls the appropriate process in your code to handle the event. For more details about event handling in Web Forms, see Web Forms Event Model.

18.36 Stages in Web Forms Processing

During Web Forms processing, the Web Forms page goes through a number of distinct stages. At each stage, the Web Forms processor calls a corresponding page processing method and runs any code in that method. These page processing methods provide you with entry points—hooks—that you can use to work with the contents of the form.

The following table lists the most commonly-used stages of page processing, when they occur, and what you can do at each stage. These stages are repeated at each round trip—that is, each time the form is requested or posted. The IsPostBack property allows you to test whether the page is being processed for the first time.

Note : There are several more stages of page processing than are listed in the following table. However, they are not used for ordinary page processing. Instead, they are primarily used by server controls on the page to initialize themselves, render themselves, and so on. If you intend to write your own Web Forms server controls, you will need to understand more about these additional stages. For details about the complete set of page processing stages, see Writing a Custom Web Forms Control.

Stage Meaning

Configuration (Page_Load Event)

Page and control view state is restored, and then the page

Page_Load event is raised. In general, read and restore values stored previously, including the following. Check (using the IsPostBack page property) whether this is the first time the page is being processed. Perform data binding the first time the page is processed, or reevaluate data binding expressions on subsequent round trips. Read and update control properties. Restore state saved from a previous client request during the Save stage.

Event handling
If the page was called in response to a form event, the corresponding event handler in the page is called during this stage. In general, perform your application-specific processing, such as the following:

Handle the specific event.

> **Note: Events are not raised in a particular order, except that cached control events (as specified by the control's AutoPostBack property) are always processed before the posting event. If the page contains validation controls, check the IsValid property for the page or for individual validation controls. Manually save the state of global page variables that you are maintaining yourself and of controls you dynamically added to the page.**

Cleanup (Page_Unload event)

The page has finished rendering and is ready to be discarded. Perform final cleanup work, such as:

- Closing files.
- Closing database connections.
- Discarding objects.

> **Note: It is important that expensive resources (like database connections) be explicitly closed. Otherwise, they will remain open until the next garbage collection occurs. On a heavily loaded server too many open resources can exhaust memory.**

18.37 Creating a Web Forms Application

18.37.1 Creating a Web Forms Project

The purpose of this chapter is to show you how to create a Web Forms application that consumes middle-tier components. You will create a Web-based application and integrate business-object components written in C#, Visual Basic, and Managed Extensions for C++.

Throughout this section, you will use a number of tools such as :

- The Web Forms Designer
- The Component Designer
- The Managed Extensions code wizards

You will also accomplish the following:

- Creating, managing, compiling, and deploying Web Forms pages.
- Creating a business object with Managed Extensions for C++.
- Creating a data-access component.
- Consuming business objects in a Web Forms application.

Real-life Web Forms applications include hotel billing, plane ticketing, or book database applications. In this chapter you are going to create a book database application as an example. However, by the end of the chapter you should be able to create any database application, where you can browse the database, select an item, and see details about that item.

For these, you need:

1. Access to a machine running Internet Information Services 5.0 with FrontPage Server Extensions installed.
2. Access to a machine running SQL Server 7.0 hosting the PUBS database. The PUBS database is one of the sample databases that ships with SQL Server 7.0.
3. Creating a Business Object with Managed Extensions for C++

A business object is used to implement business logic and business rules. For example, you can create a simple business object that gives a discount for purchases.

To explore multiple-language interoperability, implement a business object as a component in a separate project using the Managed Extensions for C++. This simulates the common situation in which a third party is building executable components that you will consume.

To create the business object, you may proceed as follows:

1. On the File menu, point to New and then click Project to open the New Project dialog box.
2. Click the Visual C++ Projects folder in the Project Type pane, then click the Managed C++ Class Library project icon in the Templates pane.
3. Name the project McPaymentRules.
4. In Class View right-click the McPaymentRules class node, point to Add, and select Add Function from the shortcut menu.
5. The Add Member Function Wizard appears.
6. In the Add Member Function Wizard, do the following:
   - Enter the name (CalcDiscount), return type (double), parameter type (double), and parameter name (price).
   - Click the Add button.
   - Click Finish to create the function.
7. To navigate to the function body in the editor pane, right-click the function name in Class View and select Go To Definition. The following code appears in the editor:

```
double CalcDiscount(double price) {
   return double();
}
```

8. Replace the return statement with:

```
   return price*0.9;
```

9. Save the project.

From the Build menu, choose Build to generate the file McPaymentRules.dll.

18.37.2 Creating the Web Application Project

In this section you create the Web Forms project that represents the main application. On the Web Form, you place your DataView to view a specific table in the database.

You are also going to add a component where you create the data-access object that will contain your data source.

To create the Web Form:

1. Create a C# or Visual Basic Web Application project. For details about creating a Web Forms project, see Creating Web Forms.
2. The Application Wizard will build the necessary project files, including the following files:
   - WebForm1.aspx — contains the visual representation of the Web Form.
   - WebForm1.cs or WebForm1.vb — contains the code for event handling and other programmatic tasks. In order to see this file in Solution Explorer, click the Show All Files icon, and then expand the WebForm1.aspx node.

To add the :

1. In Solution Explorer, right-click the Web Application project, point to Add, and click Add New Item.
2. In the Add New Item dialog box, double-click Component (or Component Class in Visual Basic) in the right pane.

This will create a new .cs or .vb file in your project. Unless you choose another name for the file, it will use the default name, Component1.

18.37.3 Creating the Data-Access Component

A dataset can be used to bind data values into the HTML transmitted to the client. It can be created either on the Web Form itself or on the component. In order to simulate real-life situations, you will create it on the component. You will also write the necessary code to fill the DataSet with data from the database.

To add and configure a data source:

1. From the Data tab of the Toolbox, drag ADODataSetCommand onto the Component Designer.

This will start the DataSetCommand Configuration Wizard.

2. In the Choose Your Data Connection page, click New Connection. In the Connection tab of the Data Link Properties dialog box:
   - Enter the Name of the server where the Pubs database is installed.
   - Enter the logon information for the server.

Tip Often when Pubs is set up as a sample database, the User name is 'sa' and you can select Blank password.

3. Select Pubs from the database list.
4. . Click OK to return to the wizard.
5. Click Next to advance to the Choose a Query Type page.

6. Select Use SQL statement, then click Next.
7. In the Generate the SQL Statements page, click SQL Builder.
8. In the Tables tab of the Add Table dialog box, click Titles, click the Add button, then click the Close button to add the Titles table.
9. Select the title_id, title, price, pubdate, and notes check boxes, then click OK.
10. The Generate the SQL Statements page displays the SQL statement that is created.
11. Click Finish to connect your data source to the Titles table and to create the following objects, which are added to the component tray:

adoDataSetCommand1 adoConnection1 Save the project.

To generate and fill the dataset:

1. Click the component designer surface and then select Generate DataSet from the DataClass menu.

The Generate DataSet Class dialog box appears.

2. Enter a name, such as myDataSet and click OK.

The myDataSet object appears in Solution Explorer.

3. Select Generate DataSetCommand Methods from the DataClass menu. This will add convenient methods to fill and update the dataset.

18.37.4 Binding the DataGrid

In this section, you add a DataView object to the Web Forms page and use it to link the DataGrid to the DataSet. After you add and configure a DataView, you will be able to select a single table (such as Titles) out of the many tables that might be within a DataSet.

To add a dataset to the form:

1. Drag DataSet from the Data tab of the Toolbox onto the form.
2. The Choose a DataSet dialog box appears.

If it is not already selected, select TypedDataSet.

3. Select the name of the dataset (YourProjectName.myDataSet) from the drop-down list.
4. This adds a dataset, myDataSet1, to your WebForms page.

To fill the dataset with data:

In the WebForms1.cs file, declare a component object at the top level in the WebForm1 class:

```
protected Component1 myComponent = new Component1();
Modify the Page_Load method (or the WebForm_Load sub) as follows:
// C#
void Page_Load (Object sender, EventArgs e) {
   if (! IsPostBack)
   {
      myComponent.FillDataSet(myDataSet1);    // Add this line
   }
}
```

Note that the statement inside the conditional block evaluates true the first time the browser hits the page.

To add and configure the DataView:

1. Switch to the Web Forms page Design view.
2. From the Data tab of the Toolbox, drag the DataView control onto the Web Forms page.
3. A new object, dataView1, is added to the tray.
4. Select the dataView1 object and press F4 to display its Properties window.
5. Open the Table property page and select myDataSet1.titles from the list.

To add and configure the DataGrid, you may proceed as follows:

1. Drag a DataGrid control from the Web Forms tab of the Toolbox onto the Web Forms page.
2. Right-click the DataGrid and select Property Builder from the shortcut menu.
3. In the General form, set the DataSource to DataView1, and the Data Key Field to the primary key of the database (title_id).
4. In the Columns form:
   - Clear Create columns automatically at runtime.
5. Scroll down in the Available Columns list until you can expand the Button Column node.
6. Expand the Button Column node and add a Select button by clicking the right arrow between the two lists.
7. Set the Header Text to 'Title', the Sort Expression to 'title' and the Text Field to 'title.'
8. Add the price column from the Available Columns list by selecting it and clicking the right arrow between the two lists.
9. Provide an appropriate header in the Header text box (such as Price).
10. Click OK.

Your DataGrid now shows only these two columns from the DataSet.

11. Switch to Code view and bind the data to the columns of the DataGrid by adding the DataBind() to Page_Load method (or the WebForm_Load sub). The method or the subroutine will look like this:

```
// C# protected
void Page_Load(object sender, EventArgs e)
{
   if (!IsPostBack)
   {
         myComponent.FillDataSet(myDataSet1); DataGrid1.DataBind();
         // Add this line
   }
}
```

To test the project

1. From the Build menu, choose Build.
2. Press F5 to start the project (or Ctrl-F5 to start without debugging).

3. You can now display data from the database on a Web Forms page and view it in your browser.

Adding a Details Panel to the Web Form:

The details panel allows you to view additional information about the selected items (books) without cluttering the DataGrid with extraneous material.

In this step, you add the second DataView, which will allow you to filter the table so that only the currently selected row is displayed.

To add a second DataView:

1. Add a second DataView to the Web Forms page.

This creates the dataView2 object.

2. Select dataView2 and press F4 to display the Properties window.

3. Expand the Data property, select Table, and then link it to myDataSet1.titles.

In this step, you add Label controls that correspond to data columns and bind each Label to the second DataView.

To add Label controls:

1. For each detail field that you are using (such as title, price, pubdate, and notes), drag a Label control from the Web Forms tab of the Toolbox onto the Web Forms page.
2. Select each Label and bind it to a field as follows:
   - Press F4 to display the Properties window of the selected label.
   - Expand the Data property.
   - Select the DataBindings property page by clicking the ellipsis button next to DataBindings.
   - Select the Text property from Bindable Properties in the left pane and select the appropriate column from the dataView2 node in the right pane to bind the Text property to the appropriate column in dataView2.
   - Select the proper format from the Format drop-down list. For example, you can use the Currency format for price and one of the available date formats for pubdate.
   - Click OK.
   - Add text to describe each Label by placing the cursor before the label's text and typing an appropriate description.
   - In this step, you add the necessary code to activate the details when you click the DataGrid.

To activate the details, you may proceed as follows:

1. Double-click the DataGrid1 object. This will add a SelectedIndexChanged event handler to your code behind and switch you to Code view.
2. Switch to the code editor view. In the DataGrid1_SelectedIndexChanged method of the code behind, set the dataView2.RowFilter so that it will select only the row you want to display:
   - myComponent.FillDataSet(myDataSet1);
   - int index = DataGrid1.SelectedIndex;
   - string key = DataGrid1.DataKeys[index].ToString();
   - dataView2.RowFilter = DataGrid1.DataKeyField + "='" + key + "'";

Conceptually, you are doing something very similar to specifying the WHERE clause of an SQL query and using this to get only a single row from the table. The DataKeys property, which was specified when you created the DataGrid, is a means for uniquely identifying each row. You determine the value of the key for the selected row by mapping the item index to a matching key. This key, which is a valid title_id, can now be used to uniquely select the row you want to use. This is true because the title_id is the primary key of the database. You cannot necessarily guarantee these constraints with a different database or with a different data key; therefore, you should alter your code if you want to use this technique on tables without a primary key.

Bind each label that you are using by calling DataBind on that label in the DataGrid1_SelectedIndexChanged method. Place these calls after the row filter change has been set:

```
// C# Label1.DataBind(); Label2.DataBind(); ...
```

Build and start the project.
You should be able to click a title and update the details with the new title's information.

Consuming the Business Object

In this step you are going to use the DLL written in Managed Extensions for C++ (McPaymentRules.dll) to calculate the discounted price for a specific book.

To consume the business object, you may proceed as follows:

1. In Solution Explorer, right-click the Web Form project and click
2. Add Reference from the shortcut menu. The Add Reference dialog box appears.
3. Select the Projects tab and click the Browse button to open the Select Component dialog box.
4. In the Select Component dialog box, click 'MyProjects' and locate the 'McPaymentRules' project folder.
5. Browse to the 'McPaymentRules.dll' file in the \bin\debug folder, select it, and click Open. The file will appear in the Select Components pane of the Add Reference dialog box. Click OK.
6. Add a new Label control (for example, Label5) and give it a name such as 'Your Price'. This label will display the price after discount.
7. View WebForms1.cs and add the following code to the end of DataGrid1_SelectedIndexChanged method:

```
// Declare an instance of the business object:
McPaymentRules pr = new McPaymentRules();
// Invoke the CalcDiscount Method:
try {
   decimal price = myDataSet1.titles[index].price;
   Label5.Text = String.Format("{0:C}",
   pr.CalcDiscount(price.ToDouble()));
}
catch {
   // If the price is blank, display a message:
```

```
        Label1.Text=" Price is not available for this item.";
        Label5.Text=" Discount is not available for this item.";
    }
```

Save all, build, and run the application.

Deploying

In this step, you deploy the project on a specific production server. To do that, switch from Debug build to Release build and rebuild the project.

To deploy the project on a production server, you may proceed as follows:

1. Right-click the solution in Solution Explorer, point to Add on the shortcut menu, and select New Project.
2. Select the Setup and Deployment Projects folder from the project type pane and Deploy Wizard from the templates pane.
3. Name the project the name that your deployed application will have on the server. For example, if your application will reside in http://ServerName/MyWebApp, name the project MyWebApp. This will bring up the Deployment Project Wizard.
4. In the Deployment Project Wizard, do the following:
   - Click 'Next' on the first wizard screen.
   - Select Deploy a web application and click Next.
   - Select Primary output from <Your project name> and Content Files from <Your project name>. Click Next.
   - If you want to add any additional files you can select them,

Click the 'Add button' and click 'Next'. Otherwise, skip this screen by clicking 'Next'.

5. Enter the remote server computer name in the text box. Select No,
6. Create the project and we will deploy it later. Click 'Next'.
7. On the summary screen, click 'Finish'.
8. In the File System Editor, right-click the Web Application Folder', point to Add and select Web Folder. Rename the new folder Bin. (If the File System Editor is not the current window, right-click the deployment project folder in Solution Explorer, point to View, and select File System.)
9. In the Web Application Folder, select all the .dll files (the project dependencies), and drag and drop them into the Bin 'folder.
10. Deploy the solution by clicking Deploy Solution on the Build menu.
11. Run your deployed solution by using the URL: "http://ServerName/MyWebApp/WebForm1.aspx."

Notice that the correct versions of all the DLLs that your application relies upon are automatically deployed to the server. This ensures that installation of other applications on this server will not break your application with incompatible file versions.

18.38 Creating Web Forms

After you have started a Web Forms project, you can create Web Forms pages.

To create a Web Forms page:

1. In Solution Explorer, right-click the name of the project, choose Add, and then choose Add New Item.
2. In the Add New Item dialog box, select Web Form in the right-hand pane.
3. Supply a name in the Name box at the bottom, and then click Open.

The new Web Forms page is displayed in the Web Forms designer. If you want to give your form a different name, close it and rename it in Solution Explorer.

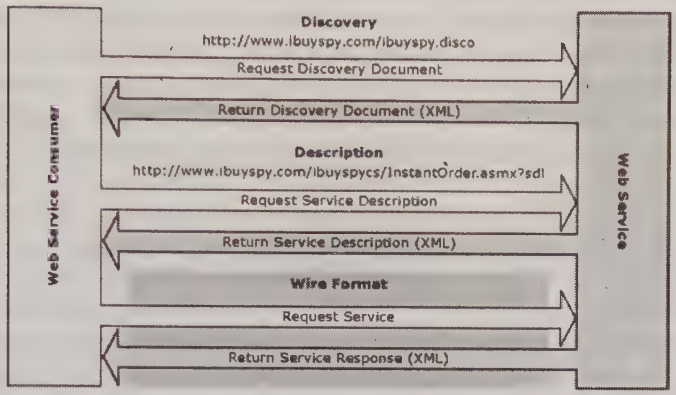
18.39 Web Services Infrastructure

Web Services are agnostic regarding the choice of operating system, object model, and programming language, which ensures their success in the heterogeneity of the Web. Web Services posses the following characteristics:

Loosely Coupled:
Web Services are considered loosely coupled since the only mandate imposed is to understand self-describing, text-based messages. A tightly coupled service, on the other hand, imposes a significant amount of customized overhead to enable communication and requires a greater understanding between the service and its clients.

Ubiquitous Communication:
It is unlikely that an operating system will be built now or in the foreseeable future that will not incorporate the ability to connect to the Internet. As such, Web Services leverage existing Internet protocols, such as XML and HTTP.

Universal Data Format:
By adopting existing, open standards instead of proprietary, closed-loop communication methods, any system supporting Internet standards can understand Web Services. XML-based messaging enables the transfer of data between autonomous and disparate systems. With these characteristics in mind, the Web Services infrastructure provides the following: a discovery mechanism to locate Web Services, a service description for defining how to use those services, and standard wire formats with which to communicate. All three characteristics exist in each piece of the infrastructure.



Infrastructure Piece Purpose

Web Service Discovery Web Services make their presence, capabilities, and supported interactions known through the process of discovery. Web Service Description To understand how to interact with a particular Web Service, it is necessary to provide a service description that defines the supported Web Service interactions. Web Service Wire Formats To enable universal communication, Web Services communicate using open wire formats, which are protocols understandable by any system capable of supporting the most common Web standards.

18.40 Visual Studio Enterprise Applications

Web Service discovery, which is a preliminary step for accessing a Web Service, is the process of locating and interrogating a Web Service. It is through the discovery process that Web Service clients learn about the existence and capabilities of a Web Service and how to properly interact with it.

Programmatic discovery can be enabled when a Web Service publishes a .disco file, which is an XML document that contains links to other resources that describe the Web Service. Discovery documents are automatically generated when creating a Web Service in Visual Studio. The following shows an example of the structure of a discovery document:

```
* <?xml version="1.0" ?> * <disco:discovery xmlns:disco="http:
//schemas.xmlsoap.org/disco" xmlns="http:
//schemas.xmlsoap.org/disco/scl"> * <scl:contractRef ref="http:
//MyWebServer/MyWebService.asmx?SDL" /> * <disco:discoveryRef
ref="SomeFolder/default.disco" />
</disco:discovery>
```

Note: The discovery document is a container for elements that typically contain links (URLs) to resources that provide discovery information for a Web Service. If the URLs are relative, they are assumed to be relative to the location of the discovery document. A Web site that implements a Web Service is not required to support discovery. Either another site could be responsible for describing the service, or there may not be a public means of finding the service, such as when the service has been created for private use only. For more information, see Discovery of Web Services (DISCO) (http://msdn.microsoft.com/xml/general/disco.asp).

The UDDI (Universal Discovery, Description and Integration) Project has created a framework for Web Service integration by providing a distributed directory of Web Services. This distributed directory enables programmatic registration and discovery of Web Services using an assortment of pre-defined SOAP messages. An UDDI compliant registry provides a service discovery platform from which to locate available Web Services within an industry or a particular company. For more information, see Universal Discovery, Description and Integration (http://www.uddi.org/).

The Web Services infrastructure is founded on communication via XML-based messages that comply with a published Web Service description. The service description is an XML document written in an XML grammar called SDL (service description language) that defines the format of messages the Web Service understands. The service description serves as an agreement that defines the behavior of a Web Service and instructs potential clients in how to interact with it. The behavior of a Web Service is determined by messaging patterns that the service defines and supports. These patterns conceptually dictate what the service consumer can expect to happen when a properly formatted message is submitted to the Web Service.

For example, the Request-Response pattern associated with a remote procedure call (RPC)-style service would define which SOAP message schema should be used for invoking a particular method. This pattern would also define the format of the ensuing response SOAP message.

Another example of a messaging pattern represents unidirectional interactions. This pattern is employed when a one-way communication is to take place. In this situation, the sender will not receive any messages from the Web Service, including fault messages.

The schemas that define the SOAP message formats can be defined internally to the actual Web Service description, or they can be defined externally and imported into the Web Service description.

In addition to message format definitions and messaging patterns, the service description also contains the address that is associated with each Web Service entry point. The format of this address will be appropriate to the protocol used to access the service, such as a URL for HTTP or an e-mail address for SMTP.

18.41 SOAP

SOAP, or Simple Object Access Protocol, is a simple, lightweight XML-based protocol for exchanging structured and type information on the Web. The overall design goal of SOAP is to keep the exchange process as simple as possible and to provide a minimum of functionality. The protocol contains no application or transport semantics. As a result, the protocol is highly modular and very extensible.

By traveling over standard Internet protocols, SOAP is able to leverage the existing open architecture of the Internet and gain easy acceptance by any arbitrary system capable of supporting the most basic Internet standards. The infrastructure required to support a SOAP-compliant Web Service could be viewed as rather simplistic, yet powerful, since it adds relatively little to the existing infrastructure of the Internet and still facilitates universal access to the services that are built upon it. SOAP is a lightweight, message-based protocol that is built on XML and standard Internet protocols, such as HTTP and SMTP. Although SOAP's roots are in RPC, it is not limited to being a request/response mechanism. Instead, SOAP has become an industry-accepted messaging infrastructure.

The SOAP protocol specification consists of two main parts. One part defines a mandatory extensible envelope for encapsulating data. The SOAP envelope defines a SOAP message and is the basic unit of exchange between SOAP message processors. Each SOAP message is a one-way transmission. SOAP messages are often combined to implement patterns such as request/response, but SOAP itself does not define a message exchange pattern.

The second part of the SOAP protocol specification defines optional data encoding rules for representing application-defined data types and directed graphs and a uniform model for serializing non-syntactic data models.

SOAP can be used in combination with any protocol that is able to transport the SOAP envelope, such as HTTP or SMTP.

Build Web Commerce Apps With C#

Use .NET and C# to quickly build a scalable commerce Web application that outperforms standard ASP Web apps.

Microsoft .NET's advanced object model enables you to quickly build secure, scalable Web applications. Combined with C#'s flexibility and power, Microsoft .NET is an ideal technology for developing commerce solutions. In this first article of a three- part series, we will use .NET and C# to construct a commerce Web application—an application that allows your customers to browse your product catalog and add items to a shopping cart.

What you need:

- Internet Explorer 5.5
- Windows 2000 Professional, Serer or Advanced Server
- ASP+/Microsoft .NET
- SQL Server 7.0 SP1 or later

The application is meant to teach you the basic techniques of .NET and C#. You will learn how to create and compile a C# business layer component. You will also see how to access the component from an ASPX page and bind data to a List control. Using these techniques, you can create secure, scalable .NET Web applications in a fraction of the time it takes to build standard ASP applications.

You must complete many tasks before you write a single line of code. Your site coding will go more quickly if you take the time to discover the requirements and plan the design. One of the most common mistakes developers make is to start coding too early.

The basis of a solid .NET application is still rooted in a solid database schema with well-written stored procedures. Once you have created the database and tested the stored procedures, develop the components your application will use to retrieve data from the database. As with standard Windows DNA Web applications and other n-tier methodologies, there are no direct calls from the presentation layer—this increases application speed and makes the app much easier to debug.

Develop the Components

You are going to build one C# component that acts as both your business logic layer and data access layer. If you did foresee a need to convert the application to another type of database such as Oracle, you would need to separate these into two components. But SQL Server is the only database you will be using, so you don't have to separate them.

This first article implements two pages: Default.aspx and ProductList.aspx. Default.aspx is the first page users see when loading your Web application. When a user clicks on a category from the menu, he or she will be taken to the ProductList.aspx screen. This page simply lists all the products in a category.

On these two pages, only two actions take place: The first is listing categories for the menu, and the second is listing products for the products list page. Perform these actions by creating two functions that each call a different stored procedure. By identifying the objects in your application up- front, you uncover the classes you will need in your component layer. The only two classes you will create are Category and Product because you're working only with the Default.aspx and ProductList.aspx pages (see Figure 1).

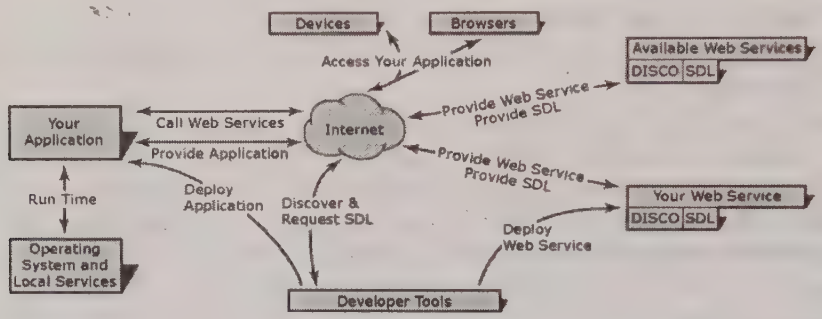


Fig. 1: You Only Need Two Classes Click here.

The two classes are organized into a single namespace called CommerceDotNet. Use CommerceDotNet.Categories to create an instance of the Categories class, and CommerceDotNet.Products to create an instance of the Products class. You should define each of these classes in separate files for cleaner and more organized code. Even though you define the classes in different files, you can still make them into a single component because they share the same namespace.

The simple CategoryList method passes back a dataset containing all the product categories in the system. Create this method by creating the Categories class and adding a method signature for CategoryList:

```
namespace CommerceDotNet {
  public class Categories {
    public DataSet CategoryList() {
    }
  }
}
```

Once you've created the class, start adding code to the CategoryList method (see Listing 1).

The CategoryList method first establishes a connection to the database and creates an instance of the Command object. It then sets the command object type to a stored procedure. Next, it executes the stored procedure and fills, and returns a dataset containing the product category data. You must import the System.Data and System.Data.SQL namespaces to use the SQLConnection, SQLDataSetCommand, and DataSet objects..

Listing 1.2: CategoryList At Work

```
using System;
using System.Data;
using System.Data.SQL;
namespace CommerceDotNet {
     public class Categories {
          public DataSet CategoryList() {
               // Create Instance of Connection
               // and Command Object
               SQLConnection myConnection = new
SQLConnection("server=localhost;uid=sa;pwd=;database=commercedotnet");
               SQLDataSetCommand myCommand = new
SQLDataSetCommand("ListCategory", myConnection);
               // Mark the Command as a SPROC
               myCommand.SelectCommand.CommandType =
CommandType.StoredProcedure;
               // Create and Fill the DataSet
               DataSet myDataSet = new DataSet();
               myCommand.FillDataSet(myDataSet, "CategoryList");
               // Return the DataSet
               return myDataSet;
          }
     }
}
```

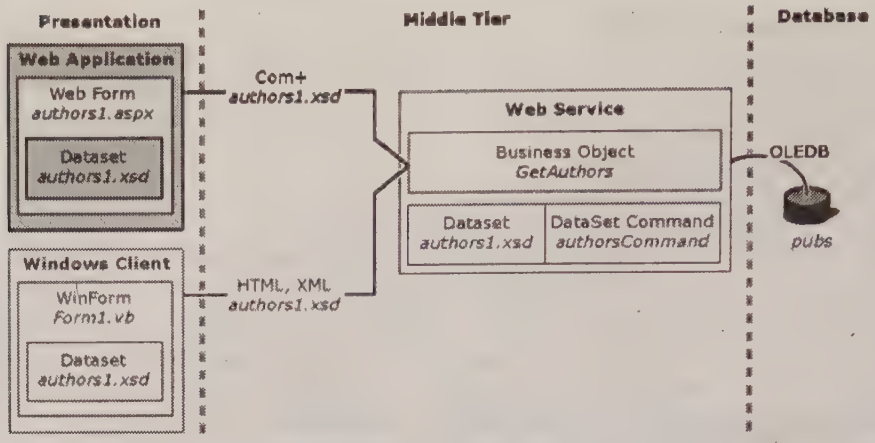
Use CategoryList to retrieve and return a dataset of all product categories in the system in four basic steps. First, create a connection to the database using the SQLConnection and SQLDataSetCommand objects. Next, mark the command as a stored procedure. Then instantiate a dataset and fill it with the results of the ListCategory stored-procedure execution. Last, return the dataset to the calling function. The ProductsList method is similar to the CategoryList method, with one difference: The ProductsList method accepts one argument that is passed to the ListProducts stored procedure as an input parameter. Because the ProductsList code resembles the CategoryList code, here is only the step of adding an input parameter to a stored procedure:

```
SQLParameter parameterCategoryID = new SQLParameter("@CategoryID",
SQLDataType.Int, 4);
parameterCategoryID.Value = categoryID;
myCommand.SelectCommand.Parameters.Add(parameterCategoryID);
```

Once you define both of these classes, compile them:

```
csc /out:../bin/CommerceDotNet.dll /t:library /r:System.Data.dll
CategoryDB.cs ProductDB.cs
```

The "/out:" switch specifies the compiled library's name and location. The "/t:" switch tells the compiler to build a library. The "/r:" switch indicates a reference to another library the component requires. At the end of the command, the two source files are listed.



Vary Your Formatting

Develop the ASP+ Pages

Once you create and test the components, implement the ASP+ pages that will function as the applications user interface. In the first of the three articles, we'll implement only the default page and the products list page. You start by creating the Default.aspx page, the first page you see when loading the site.

The main item of interest on the Default.aspx page is the category list on the left-hand side (see above figure). We created this using Web forms, specifically the DataList control. DataList is a server-side control you instantiate by using the

<asp:...></asp> tags in the same manner as all Web controls

Listing : Create the Default-Page Menu

Use the DataList server-side control to create the menu on the Default.aspx page. Two templates are defined for the DataList: ItemTemplate and SelectedItemTemplate. Each of the items in the =DataList uses ItemTemplate to determine its formatting, with the exception of the selected item, which uses SelectedItemTemplate.

```
<asp:DataList id="MyList" runat="server" cellpadding="3"
cellspacing="0" width="145" SelectedItemStyle-BackColor="dimgray"
SelectedItemStyle-CssClass="dimgray" maintainstate="false">
<template name="ItemTemplate">
<asp:HyperLink class="MenuText" id=HyperLink1 Text='<%#
DataBinder.Eval(Container.DataItem, "Name") %>' NavigateUrl='<%#
"productlist.aspx?CategoryID=" + DataBinder.Eval(Container.DataItem,
"CategoryID") %>' runat="server" />
</template>
<template name="SelectedItemTemplate">
<asp:HyperLink class="MenuText" id=HyperLink2 Text='<%#
```

```
DataBinder.Eval(Container.DataItem, "Name") %>' NavigateUrl='<%#
"productlist.aspx?CategoryID=" + DataBinder.Eval(Container.DataItem,
"CategoryID") %>' runat="server" />
</template>
</asp:DataList>
```

Format the DataList content using nested <template></template> blocks code inside the DataList. You can choose from several templates, but this menu requires only two: ItemTemplate and SelectedItemTemplate. Each of the items in the DataList is formatted according to the ItemTemplate, with the exception of the selected item, which is formatted by the SelectedItemTemplate.

At first sight, the template code may appear daunting. However, consider the goal: You want to create a list of category hyperlinks that will take the users to the correct product listing for that category. So, inside the template tag, you'll notice a hyperlink Web control. Insert the CategoryID as part of the query string to create the hyperlink, using the DataBinder object and the syntax in Listing 2. The DataBinder object formats and inserts the field of data into your HTML.

Note that you didn't need to use a Web control inside the template. Instead of:

```
<asp:HyperLink.../>
```

it is perfectly acceptable to use:

```
<a href="<%# "productlist.aspx?CategoryID=" +
```

The next step: Bind the information retrieved from the database to the DataList when the page loads. First, declare a variable named menuItems as an ICollection (the datatype returned from the component layer). Next, instantiate the component, call the CategoryList method, and store the dataset in the menuItems object. The DataList's data source is then set to the collection of data. MyList.DataBind enables the control to iterate through the dataset (see Listing 3).

Listing 3 : Bind a DataList to a Data Source

This C# code in the Page_Load event binds the DataList to a data source. First, it retrieves the data and places it in a dataset (ICollection object). Second, it sets the DataList's DataSource property to that dataset, and then runs the DataBind method to cause the DataList to iterate through the dataset.

```
<script language="C#" runat="server"> void Page_Load(Object sender,
EventArgs e) {
          ICollection menuItems;
CommerceDotNet.Categories categories = new CommerceDotNet.Categories();
        menuItems = categories.CategoryList().Tables[0].DefaultView;
          // Associate the list of menu items
          // into the list
          //   and bind to the list
          MyList.DataSource = menuItems;
          MyList.DataBind();
    }
</script>
```

Next, create the ProductList.aspx page, which generates the list of products in a given category once the user chooses a category. The DataList on the ProductList.aspx page introduces the AlternatingItemTemplate, which allows to change the appearance of alternating items in the DataList. This makes data much more readable. Note how the background color of alternating items changes (see Figure in page 610).

You can easily control the data's appearance with the DataList by using its built-in properties. The RepeatDirection property allows you to display the products vertically or horizontally in the column. Similarly, the RepeatColumns property allows you to control the number of columns the list displays (see Figure in page 610). To get this kind of functionality in standard ASP, you'd need to create a loop and create the table manually while iterating through the recordset.

The ability to format the DataList Web control's output this way gives the Web developer a lot of power and flexibility by offering respite from worrying about minor UI nuances. You could even make these settings user-configurable options; you'll learn this in the third part of this article series.

These are the initial steps for building your own commerce solution in Microsoft .NET and C#. Recall that Microsoft .NET and C# serve you particularly well here because they allow you to create a stable, scalable Web application in fewer lines of code. We also started discussing component and Web-form development, including the applications main page and a page to display a list of products. In the next article, we'll build more advanced Web forms as well as add more functionality to the business layer components. You will also learn how to add user authentication to a Web application.

18.42 Sending an SMTP Email in HTML Format

This section shows you how to send an HTML document as an SMTP email using SmtpMail.Send(MailMessage);.

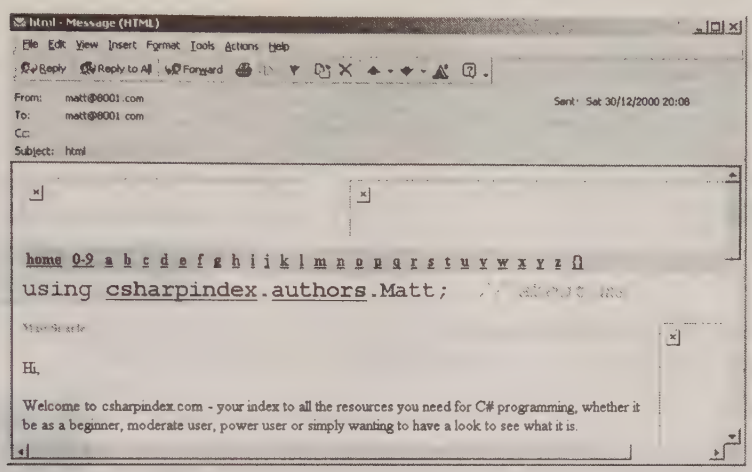
By default SmtpMail.Send sends email messages in text format. In order to send an email in HTML format we need to add the following line to our code before sending the mail:

```
mail.BodyFormat = MailFormat.Html;
```

That's all you need to do - Now just send a valid HTML formatted document as the body of the message.

If you're using the code from the previous article then you'll receive an exception/error if you don't also attach a file. The code needs to check if there is an attachment specified and if so - whether it is valid - So either specify a valid file to attach or comment out the line of code that attaches the file. - You may need to add a <BASE> HTML tag as explained here:

The <BASE> HTML Tag If we were to send the HTML code from http://csharpindex.com/authors/Matt/ the user would be delivered something like the following message:


html - Message (HTML)
File Edit View Insert Format Tools Actions Help
Reply Reply to All Forward
From: matt@8001.com
To: matt@8001.com
Cc:
Subject: html
Sent: Sat 30/12/2000 20:08
home 0-9 a b c d e f g h i j k l m n o p q r s t u v w x y z Ω
using csharpindex.authors.Matt;
Hi,
Welcome to csharpindex.com - your index to all the resources you need for C# programming, whether it be as a beginner, moderate user, power user or simply wanting to have a look to see what it is.

The images, style sheets, java scripts and other included files are not loaded, so all the recipient gets is the text content and invalid links. The reason for this is because the links in the page do not explicitly specify http://csharpindex.com as the base URL. i.e. the link to the logo (top left image) is pointing to /images/csharpindexLogo.jpg.

So we have two choices:

1. Change all links to include http://csharpindex.com

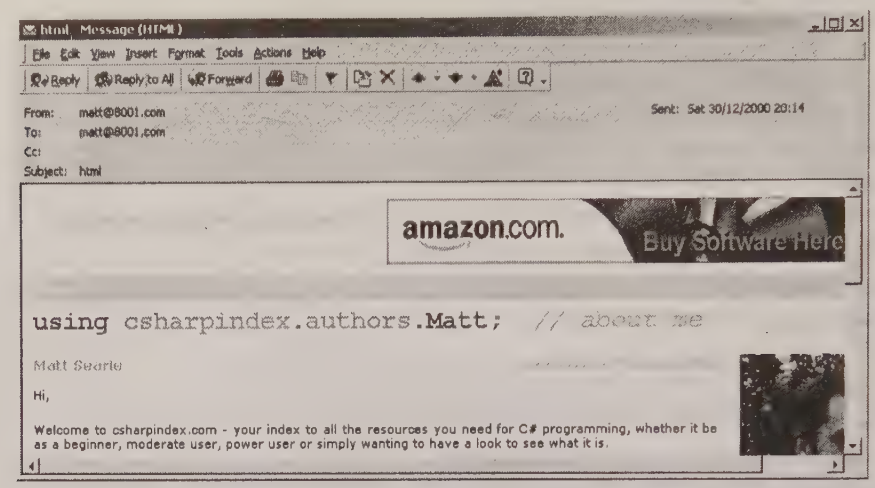
```
<HTML>
<BODY>
  <A HREF="http://csharpindex.com/code/csharp.asp">C# Code
Samples</A>
  <A HREF="http://csharpindex.com/colorCode/">colorCode C# code
to HTML</A>
  <A HREF="http://csharpindex.com/news/">News and Articles</A>
</BODY>
</HTML>
csharpindex.com/colorCode
```

or,

2. Add the <BASE> HTML tag to the HTML <HEAD> section.

```
<HTML>
<HEAD>
  <BASE HREF="http://csharpindex.com/authors/Matt/">
</HEAD>
<BODY>
  <A HREF="/code/csharp.asp">C# Code Samples</A>
  <A HREF="/colorCode/">colorCode C# code to HTML</A>
  <A HREF="/news/">News and Articles</A>
</BODY>
</HTML>
csharpindex.com/colorCode
```

Now sending the email would be received as:


html Message (HTML)
File Edit View Insert Format Tools Actions Help
Reply Reply to All Forward
From: matt@8001.com
To: matt@8001.com
Cc:
Subject: html
Sent: Sat 30/12/2000 20:14
amazon.com.
Buy Software Here
using csharpindex.authors.Matt; // about me
Matt Searle
Hi,
Welcome to csharpindex.com - your index to all the resources you need for C# programming, whether it be as a beginner, moderate user, power user or simply wanting to have a look to see what it is.

TIP: Although using <BASE href=http://csharpindex.com> will work in most cases, it is good practice to use the full path to the document: <BASE href=http://csharpindex.com/authors/Matt/> in case you have links like ../photo.jpg in the HTML code.

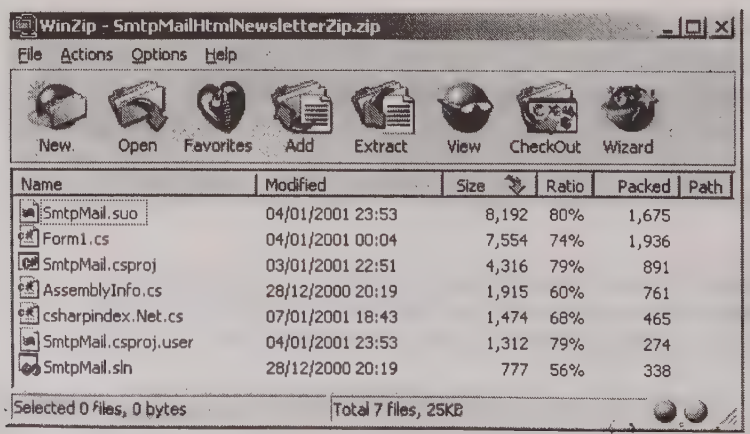
You don't have to use the HTML code from an existing web page. You can of course construct the HTML document in your C# code or by typing the HTML document into the body text box.

In the next article we will change the application to fetch the page, add the <BASE> HTML tag (if it doesn't already exist) and send it.

Requirements:

```
Namespace: System.Web.Util;
Assembly: System.Web.DLL
```

The Code


WinZip - SmtpMailHtmlNewsletterZip.zip
File Actions Options Help
New. Open Favorites Add Extract View CheckOut Wizard
Name Modified Size Ratio Packed Path
SmtpMail.suo 04/01/2001 23:53 8,192 80% 1,675
Form1.cs 04/01/2001 00:04 7,554 74% 1,936
SmtpMail.csproj 03/01/2001 22:51 4,316 79% 891
AssemblyInfo.cs 28/12/2000 20:19 1,915 60% 761
csharpindex.Net.cs 07/01/2001 18:43 1,474 68% 465
SmtpMail.csproj.user 04/01/2001 23:53 1,312 79% 274
SmtpMail.sln 28/12/2000 20:19 777 56% 338
Selected 0 files, 0 bytes
Total 7 files, 25KB

```
//Form1.cs
namespace SmtpMail {
    using System;
    using System.Drawing;
    using System.Collections;
    using System.ComponentModel;
    using System.WinForms;
    using System.Data;
    using System.Web.Util;

    /// <summary>
    ///    Summary description for Form1.
    /// </summary>
    public class Form1 : System.WinForms.Form
    {
        /// <summary>
        ///    Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components;
        private System.WinForms.Label lblAttach;
        private System.WinForms.TextBox txtAttachments;
        private System.WinForms.TextBox txtMessage;
        private System.WinForms.Label lblMessage;
        private System.WinForms.Label lblFrom;
        private System.WinForms.TextBox txtFrom;
        private System.WinForms.Label lblSubject;
        private System.WinForms.TextBox txtSubject;
        private System.WinForms.Label lblBcc;
        private System.WinForms.TextBox txtBcc;
        private System.WinForms.Label lblCc;
        private System.WinForms.TextBox txtCc;
        private System.WinForms.Label lblTo;
        private System.WinForms.Button cmdSend;
        private System.WinForms.TextBox txtTo;
        public Form1()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();
            //
            // TODO: Add any constructor code after
                //InitializeComponent call
            //
        }
        /// <summary>
```

```
///    Clean up any resources being used.
/// </summary>
public override void Dispose()
{
    base.Dispose();
    components.Dispose();
}
/// <summary>
///    Required method for Designer support - do not
          //modify
///    the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.lblBcc = new System.WinForms.Label ();
    this.txtBcc = new System.WinForms.TextBox ();
    this.txtTo = new System.WinForms.TextBox ();
    this.lblFrom = new System.WinForms.Label ();
    this.lblMessage = new System.WinForms.Label ();
    this.lblTo = new System.WinForms.Label ();
    this.txtSubject = new System.WinForms.TextBox ();
    this.txtFrom = new System.WinForms.TextBox ();
    this.txtAttachments = new System.WinForms.TextBox ();
    this.lblAttach = new System.WinForms.Label ();
    this.txtCc = new System.WinForms.TextBox ();
    this.cmdSend = new System.WinForms.Button ();
    this.lblCc = new System.WinForms.Label ();
    this.txtMessage = new System.WinForms.TextBox ();
    this.lblSubject = new System.WinForms.Label ();
    //@this.TrayHeight = 0;
    //@this.TrayLargeIcon = false;
    //@this.TrayAutoArrange = true;
    lblBcc.Location = new System.Drawing.Point (8, 152);
    lblBcc.Text = "&Bcc:";
    lblBcc.Size = new System.Drawing.Size (25, 13);
    lblBcc.AutoSize = true;
    lblBcc.TabIndex = 6;
    txtBcc.Location = new System.Drawing.Point (64, 144);
    txtBcc.TabIndex = 7;
    txtBcc.Size = new System.Drawing.Size (320, 20);
    txtTo.Location = new System.Drawing.Point (64, 80);
    txtTo.TabIndex = 3;
    txtTo.Size = new System.Drawing.Size (320, 20);
    lblFrom.Location = new System.Drawing.Point (8, 56);
```

```
            lblFrom.Text = "&From:";
            lblFrom.Size = new System.Drawing.Size (33, 13);
            lblFrom.AutoSize = true;
            lblFrom.TabIndex = 0;
            lblMessage.Location = new System.Drawing.Point (8,256);
            lblMessage.Text = "&Message:";
            lblMessage.Size = new System.Drawing.Size (52, 13);
            lblMessage.AutoSize = true;
            lblMessage.TabIndex = 12;
            lblTo.Location = new System.Drawing.Point (8, 88);
            lblTo.Text = "&To:";
            lblTo.Size = new System.Drawing.Size (20, 13);
            lblTo.AutoSize = true;
            lblTo.TabIndex = 2;
            txtSubject.Location = new System.Drawing.Point (64,
176);
            txtSubject.TabIndex = 9;
            txtSubject.Size = new System.Drawing.Size (320, 20);
            txtFrom.Location = new System.Drawing.Point (64, 48);
            txtFrom.TabIndex = 1;
            txtFrom.Size = new System.Drawing.Size (320, 20);
            txtAttachments.Location = new System.Drawing.Point
(64, 208);
            txtAttachments.TabIndex = 11;
            txtAttachments.Size = new System.Drawing.Size (320,
20);
            lblAttach.Location = new System.Drawing.Point (8,
216);
            lblAttach.Text = "&Attach:";
            lblAttach.Size = new System.Drawing.Size (38, 13);
            lblAttach.AutoSize = true;
            lblAttach.TabIndex = 10;
            txtCc.Location = new System.Drawing.Point (64, 112);
            txtCc.TabIndex = 5;
            txtCc.Size = new System.Drawing.Size (320, 20);
            cmdSend.Location = new System.Drawing.Point (8, 8);
            cmdSend.Size = new System.Drawing.Size (75, 23);
            cmdSend.TabIndex = 14;
            cmdSend.Text = "&Send";
            cmdSend.Click += new System.EventHandler
(this.cmdSend_Click);
            lblCc.Location = new System.Drawing.Point (8, 120);
            lblCc.Text = "&Cc:";
            lblCc.Size = new System.Drawing.Size (21, 13);
            lblCc.AutoSize = true;
```

```
            lblCc.TabIndex = 4;
            txtMessage.Location = new System.Drawing.Point (8,
272);
            txtMessage.Multiline = true;
            txtMessage.TabIndex = 13;
            txtMessage.Size = new System.Drawing.Size (376, 184);
            lblSubject.Location = new System.Drawing.Point (8,
184);
            lblSubject.Text = "Sub&ject:";
            lblSubject.Size = new System.Drawing.Size (44, 13);
            lblSubject.AutoSize = true;
            lblSubject.TabIndex = 8;
            this.Text = "SmtpMail";
            this.AutoScaleBaseSize = new System.Drawing.Size (5,
13);
            this.ClientSize = new System.Drawing.Size (392, 461);
            this.Controls.Add (this.lblAttach);
            this.Controls.Add (this.txtAttachments);
            this.Controls.Add (this.txtMessage);
            this.Controls.Add (this.lblMessage);
            this.Controls.Add (this.lblFrom);
            this.Controls.Add (this.txtFrom);
            this.Controls.Add (this.lblSubject);
            this.Controls.Add (this.txtSubject);
            this.Controls.Add (this.lblBcc);
            this.Controls.Add (this.txtBcc);
            this.Controls.Add (this.lblCc);
            this.Controls.Add (this.txtCc);
            this.Controls.Add (this.lblTo);
            this.Controls.Add (this.cmdSend);
            this.Controls.Add (this.txtTo);
        }
        protected void cmdSend_Click (object sender,  System.EventArgs
e)
        {
            MailMessage mail = new MailMessage();
            mail.From = txtFrom.Text;
            mail.To = txtTo.Text;
            mail.Cc = txtCc.Text;
            mail.Bcc = txtBcc.Text;
            mail.Subject = txtSubject.Text;
            mail.BodyFormat = MailFormat.Html;
            mail.Body = txtMessage.Text;
            // Add an attachment
            mail.Attachments.Add(new
```

```
MailAttachment(txtAttachments.Text));
            SmtpMail.Send(mail);
                    MessageBox.Show("Message  Sent.",  "SmtpMail"
MessageBox.IconInformation | MessageBox.OK);

        }
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        public static void Main(string[] args)
        {
            Application.Run(new Form1());
        }
    }
}
```

18.43 Sending an HTML Web Page as an SMTP Email

This sample code shows you how to send any valid HTTP:// web page as an SMTP email in HTML format.

This program is useful in several ways: - Sending better looking newsletters to subscribers of your news articles from an existing page on your web site. - Sending yourself a copy of a web page, rather than using copy and paste in Internet Explorer. - Sending a friend a copy of web page you think may be of interest to them, rather than the URL - especially if the page is updated frequently and your recipients might not read their mail quickly enough to read the details of the dynamic web page.

What's going on?

As in the previous section we're sending the body of the message in HTML format. However, rather than typing or copying the HTML code into the message body we simply specify the URL to an existing web page.

This program requests the specified web page using an instance of the WebRequestFactory class. We've run off and fetched existing web pages before: see CheapISBN version 1.00 console program and CheapISBN version 1.01 ASP.NET application to fetch web pages from on-line bookstores.

For instance,if you frequently use a class called ReadUrl you created, you can move to a seperate file.

```
//  csharpindex.Net.cs
namespace csharpindex.Net
{
    using System;
    using System.IO;
    using System.Net;
    /// <summary>
    ///    Read URL to return html string
```

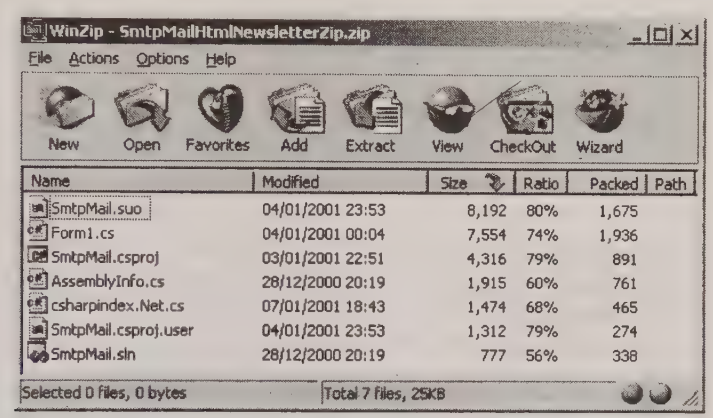
```
    /// </summary>
    public class ReadUrl
    {
        public ReadUrl(string url)
        {
            this.url = url;
        }
        public ReadUrl(string url, string username, string password)
        {
            this.url = url;
            this.username = username;
            this.password = password;
        }
        public ReadUrl(string url, string username, string  password,
string domain)
        {
            this.url = url;
            this.username = username;
            this.password = password;
            this.domain = domain;
        }
        string url;
        string username;
        string password;
        string domain;
        public string Html
        {
            get
            {
                URI uri = new URI(url);
                // Create the request object
                WebRequest req = WebRequestFactory.Create(uri);
                // Set username, password & domain
                if (domain != "")
                    req.Credentials = new
SingleCredential(username, password);
                else if (username != "")
                    req.Credentials = new
SingleCredential(username, password, domain);
                WebResponse resp = req.GetResponse();
                Stream stream = resp.GetResponseStream();
                StreamReader sr = new StreamReader(stream);
                string s = sr.ReadToEnd();
                return(s);
            }
        }
    }
}
```

ReadUrl now has three overloaded constructors. If you need a username and password to logon then the newsletter will not work as this is only used by the program requesting the HTML content to send out as the body of the email (unless of course your recipients have a valid username and password to enter when prompted).

All graphics and appropriate links are retained as the HTML code is sent as the message body and the <BASE> HTML tag inserted if it doesn't already exist on the web page. See the previous article for an explaination of why the <BASE> tag is required.

Requirements:

```
Namespace: System.Web.Util;
Assembly: System.Web.DLL
```


WinZip - SmtpMailHtmlNewsletterZip.zip
File Actions Options Help
New Open Favorites Add Extract View CheckOut Wizard
Name Modified Size Ratio Packed Path
SmtpMail.suo 04/01/2001 23:53 8,192 80% 1,675
Form1.cs 04/01/2001 00:04 7,554 74% 1,936
SmtpMail.csproj 03/01/2001 22:51 4,316 79% 891
AssemblyInfo.cs 28/12/2000 20:19 1,915 60% 761
csharpindex.Net.cs 07/01/2001 18:43 1,474 68% 465
SmtpMail.csproj.user 04/01/2001 23:53 1,312 79% 274
SmtpMail.sln 28/12/2000 20:19 777 56% 338
Selected 0 files, 0 bytes
Total 7 files, 25KB

7The Code

```
// Form1.cs
    namespace SmtpMail {
    using csharpindex.Net;
    using System;
    using System.Drawing;
    using System.Collections;
    using System.ComponentModel;
    using System.WinForms;
    using System.Data;
    using System.Web.Util;
    /// <summary>
    /// Send an SmtpMail message from a web page in HTML format.
    /// </summary>
    public class Form1 : System.WinForms.Form
    {
        /// <summary>
        ///    Required designer variable.
        /// </summary>
        private System.ComponentModel.Container components;
        private System.WinForms.TextBox txtURL;
        private System.WinForms.Label lblMessage;
```

```
        private System.WinForms.Label lblFrom;
        private System.WinForms.TextBox txtFrom;
        private System.WinForms.Label lblSubject;
        private System.WinForms.TextBox txtSubject;
        private System.WinForms.Label lblBcc;
        private System.WinForms.TextBox txtBcc;
        private System.WinForms.Label lblCc;
        private System.WinForms.TextBox txtCc;
        private System.WinForms.Label lblTo;
        private System.WinForms.Button cmdSend;
        private System.WinForms.TextBox txtTo;
        public Form1()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();
          // // TODO:Add any constructor code after InitializeComponent
call
            //
        }
        /// <summary>
        ///    Clean up any resources being used.
        /// </summary>
        public override void Dispose()
        {
            base.Dispose();
            components.Dispose();
        }
        /// <summary>
        ///    Required method for Designer support -
         //do not modify
        ///    the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.txtSubject = new System.WinForms.TextBox ();
            this.txtBcc = new System.WinForms.TextBox ();
            this.lblBcc = new System.WinForms.Label ();
            this.lblTo = new System.WinForms.Label ();
            this.lblMessage = new System.WinForms.Label ();
            this.txtURL = new System.WinForms.TextBox ();
            this.cmdSend = new System.WinForms.Button ();
            this.txtFrom = new System.WinForms.TextBox ();
            this.txtCc = new System.WinForms.TextBox ();
            this.lblSubject = new System.WinForms.Label ();
            this.lblFrom = new System.WinForms.Label ();
            this.txtTo = new System.WinForms.TextBox ();
```

```
            this.lblCc = new System.WinForms.Label ();
            //@this.TrayHeight = 0;
            //@this.TrayLargeIcon = false;
            //@this.TrayAutoArrange = true;
            txtSubject.Location = new System.Drawing.Point (64,176);
            txtSubject.Text = "C# Books";
            txtSubject.TabIndex = 9;
            txtSubject.Size = new System.Drawing.Size (320, 20);
            txtBcc.Location = new System.Drawing.Point (64, 144);
            txtBcc.TabIndex = 7;
            txtBcc.Size = new System.Drawing.Size (320, 20);
            lblBcc.Location = new System.Drawing.Point (8, 152);
            lblBcc.Text = "&Bcc:";
            lblBcc.Size = new System.Drawing.Size (25, 13);
            lblBcc.AutoSize = true;
            lblBcc.TabIndex = 6;
            lblTo.Location = new System.Drawing.Point (8, 88);
            lblTo.Text = "&To:";
            lblTo.Size = new System.Drawing.Size (20, 13);
            lblTo.AutoSize = true;
            lblTo.TabIndex = 2;
            lblMessage.Location = new System.Drawing.Point (8, 216);
            lblMessage.Text = "&URL:";
            lblMessage.Size = new System.Drawing.Size (29, 13);
            lblMessage.AutoSize = true;
            lblMessage.TabIndex = 12;
            txtURL.Location = new System.Drawing.Point (64, 208);
            txtURL.Text =
"http://csharpindex.com/books/csharp.asp";
            txtURL.TabIndex = 15;
            txtURL.Size = new System.Drawing.Size (320, 20);
            cmdSend.Location = new System.Drawing.Point (8, 8);
            cmdSend.Size = new System.Drawing.Size (75, 23);
            cmdSend.TabIndex = 14;
            cmdSend.Text = "&Send";
                                    cmdSend.Click   +=   new
System.EventHandler(this.cmdSend_Click);
            txtFrom.Location = new System.Drawing.Point (64, 48);
            txtFrom.Text = "\""My Name\" <myemail@address.com>";
            txtFrom.TabIndex = 1;
            txtFrom.Size = new System.Drawing.Size (320, 20);
            txtCc.Location = new System.Drawing.Point (64, 112);
            txtCc.TabIndex = 5;
            txtCc.Size = new System.Drawing.Size (320, 20);
            lblSubject.Location = new System.Drawing.Point (8,184);
            lblSubject.Text = "Sub&ject:";
            lblSubject.Size = new System.Drawing.Size (44, 13);
            lblSubject.AutoSize = true;
            lblSubject.TabIndex = 8;
            lblFrom.Location = new System.Drawing.Point (8, 56);
```

```
            lblFrom.Text = "&From:";
            lblFrom.Size = new System.Drawing.Size (33, 13);
            lblFrom.AutoSize = true;
            lblFrom.TabIndex = 0;
            txtTo.Location = new System.Drawing.Point (64, 80);
            txtTo.TabIndex = 3;
            txtTo.Size = new System.Drawing.Size (320, 20);
            lblCc.Location = new System.Drawing.Point (8, 120);
            lblCc.Text = "&Cc:";
            lblCc.Size = new System.Drawing.Size (21, 13);
            lblCc.AutoSize = true;
            lblCc.TabIndex = 4;
            this.Text = "SmtpMail";
            this.StartPosition =
System.WinForms.FormStartPosition.CenterScreen;
            this.AutoScaleBaseSize = new System.Drawing.Size (5,
13);
            this.ClientSize = new System.Drawing.Size (392, 245);
            this.Controls.Add (this.txtURL);
            this.Controls.Add (this.lblMessage);
            this.Controls.Add (this.lblFrom);
            this.Controls.Add (this.txtFrom);
            this.Controls.Add (this.lblSubject);
            this.Controls.Add (this.txtSubject);
            this.Controls.Add (this.lblBcc);
            this.Controls.Add (this.txtBcc);
            this.Controls.Add (this.lblCc);
            this.Controls.Add (this.txtCc);
            this.Controls.Add (this.lblTo);
            this.Controls.Add (this.cmdSend);
            this.Controls.Add (this.txtTo);
        }
        protected void cmdSend_Click (object sender,  System.EventArgs
e)
        {
            MailMessage mail = new MailMessage();
            mail.From = txtFrom.Text;
            mail.To = txtTo.Text;
            mail.Cc = txtCc.Text;
            mail.Bcc = txtBcc.Text;
            mail.Subject = txtSubject.Text;
            mail.BodyFormat = MailFormat.Html;
            // Get html doc at URL specified
            ReadUrl webPage = new ReadUrl(txtURL.Text);
            mail.Body = webPage.Html;
            // Ensure there is a <BASE> HTML tag - if not insert  one
            // Check if directory (ends in "/") then this is the
               //base else get
            // http://csharpindex.com/books/ from
```

```
                //http://csharpindex.com/books/default.asp
            string baseHref;
            int insertPos;
            if (mail.Body.ToLower().IndexOf("<base") < 1)
            {
                // Build base tag
                baseHref = txtURL.Text;
                if (baseHref.EndsWith("/") == false)
                {
                    // Take left of and including the last "/"
                    baseHref = baseHref.Substring(0,
baseHref.LastIndexOf("/") + 1);
                }
                baseHref = "<BASE HREF=" + baseHref + ">";
                // Web page does not include a <BASE tag - insert  one
after                       //<HEAD> if there is one
                insertPos = mail.Body.ToLower().IndexOf("<head>")+ 6;
                if (insertPos < 1)
                {
                    // Doesn't include a <HEAD> tag - insert this
too
                    insertPos =
mail.Body.ToLower().IndexOf("<html>") + 6;
                    baseHref = "<HEAD>" + baseHref + "</HEAD>";
                }
                // Insert it
                mail.Body = mail.Body.Insert(insertPos,
baseHref);
            }
            SmtpMail.Send(mail);
                  MessageBox.Show("Message  Sent.",  "SmtpMail",
MessageBox.IconInformation | MessageBox.OK);
        }
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        public static void Main(string[] args)
        {
            Application.Run(new Form1());
        }
    }
}
 csharpindex.com/colorCode
```

19

UNSAFE CODE AND SECURITY

19.1 Unmanaged Code

C# allows you to write unmanaged code ie; code using pointers—in cases where performance is critical or for other situations where pointers are required.

Note that unmanaged code is not verifiable by the Common Language Runtime and can only be executed when trusted.

The following example shows the use of pointers to copy an array of bytes from src to dst.

Listing 19.1

```
000: // Unsafe\fastcopy.cs
001: using System;
002:
003: class Test
004: {
005:     static unsafe void Copy(byte[] src, int srcIndex,
006:                             byte[] dst, int dstIndex, int
     count)
007:     {
008:         if (src == null || srcIndex < 0 ||
009:             dst == null || dstIndex < 0 || count < 0)
010:         {
011:             throw new ArgumentException();
012:         }
013:         int srcLen = src.Length;
014:         int dstLen = dst.Length;
015:         if (srcLen - srcIndex < count ||
016:             dstLen - dstIndex < count)
017:         {
018:             throw new ArgumentException();
019:         }
020:         fixed (byte* pSrc = src, pDst = dst)
```

```
021:         {
022:             byte* ps = pSrc;
023:             byte* pd = pDst;
024:             for (int n = count >> 2; n != 0; n—)
025:             {
026:                 *((int*)pd) = *((int*)ps);
027:                 pd += 4;
028:                 ps += 4;
029:             }
030:             for (count &= 3; count != 0; count—)
031:             {
032:                 *pd = *ps;
033:                 pd++; 034: ps++;
035:             }
036:         }
037:     }
038:
039:     static void Main(string[] args)
040:     {
041:         byte[] a = new byte[100];
042:         byte[] b = new byte[100];
043:         for(int i=0; i<100; ++i)
044:           a[i] = (byte)i;
045:         Copy(a, 0, b, 0, 100);
046:         Console.WriteLine("The first 10 elements are:");
047:         for(int i=0; i<10; ++i)
048:           Console.Write(b[i]+" ");
049:     }
050: }
```

The output of this program will be displayed as shown below:

```
C:\visual>fastcopy
The first 10 elements are:
0 1 2 3 4 5 6 7 8 9
C:\visual>
```

Explanation

Line 005 contains the unsafe keyword, which allows pointers to be used within the function. Line 020 uses the fixed statement to declare pointers to the source and destination arrays. The fixed statement 'pins' the location of the src and dst objects in memory so that they will not be moved by garbage collection. The objects will be unpinned when the fixed block completes Line 024 loops over the count in blocks of 4 bytes, copying an integer at a time. Remember that int in C# is strictly a 4-byte type. Line 030 completes the copy by moving any bytes that weren't moved in blocks of 4.

Compile with the /unsafe option.

19.2 The COMImport attribute

[This topic is part of a beta release and is subject to change in future releases. Blank topics are included as placeholders.]

When placed on a class, the COMImport attribute marks the class as an externally implemented COM class. Such a class declaration enables the use of a C# name to refer to a COM class.

```
namespace System.Runtime.InteropServices {
    [AttributeUsage(AttributeTargets.Class)]
    public class COMImportAttribute: System.Attribute
    {
        public COMImportAttribute() {_}
    }
}
```

A class that is decorated with the COMImport attribute is subject to the following restrictions:

1. It must also be decorated with the Guid attribute, which specifies the CLSID for the COM class being imported. A compile-time error occurs if a class declaration includes the COMImport attribute but without the Guid attribute.
2. It must not have any members. (A public constructor with no parameters is automatically provided.)
3. It must not derive from a class other than object.

Listing 19.2

```
using System.Runtime.InteropServices; [COMImport, Guid("00020810-0000-
0000-C000-000000000046")] class Worksheet {}
class Test {
    static void Main() {
        Worksheet w = new Worksheet();    // Creates an Excel worksheet
    }
}
```

declares a class Worksheet as a class imported from COM that has a CLSID of "00020810-0000-0000-C000-000000000046". Instantiating a Worksheet instance causes a corresponding COM instantiation.

Unsafe

The unsafe keyword denotes an unsafe context. Unsafe context is required for any operation involving pointers.

unsafe can be applied as a modifier in the declaration of callable members such as methods, properties, constructors, and extenders (but not static constructors).

```
static unsafe void FastCopy ( byte[] src, byte[] dst, int count ) {
        // unsafe context: can use pointers here
}
```

The scope of the unsafe context extends from the parameter list to the end of the function, so pointers can also be used in the parameter list:

```
static unsafe void FastCopy ( byte* ps, byte* pd, int count )
{...}
```

Alternatively, unsafe can be used to specify an unsafe statement a statement or block that is an unsafe context:

```
unsafe *p++ = *q++;     // unsafe assignment statement
unsafe { *p++ = 1; *q++ = 2; }  // unsafe block
 unsafe fixed (byte* p = src, q = dst) {...}  // unsafe fixed
statement
```

To compile unmanaged code you must specify the /unsafe compiler option. Unmanaged code is not verifiable by the Common Language Runtime.

Listing 19.3

```
using System;
class UnsafeTest {
   // unsafe method: takes pointer to int
   unsafe static void SquarePtrParam (int* p)
   {
      *p *= *p;
   }
   public static void Main()
   {
      int i = 5;
      // unsafe statement for address-of operator
      unsafe SquarePtrParam (&i);
      Console.WriteLine (i);
   }
}
```

The output of this program shall get displayed as shown below:

```
C:\visual>UnsafeTest
25
```

19.3 Fixed

Prevents relocation of a variable by the garbage collector.

fixed (type* ptr = expr) statement where:

type
An unmanaged type or void.

ptr
A pointer name.

expr

An expression that is implicitly convertible to type*.

statement
Executable statement or block.

Remarks
The fixed statement is only permitted in an unsafe context.

The fixed statement sets a pointer to a managed variable and "pins" that variable during the execution of statement. Without this statement, pointers to managed variables would be of little use since garbage collection could relocate the variables unpredictably. (In fact, the C# compiler will not allow you to set a pointer to a managed variable except in a fixed statement.)

```
// assume
class Point
{
   public int x, y;
}
Point pt = new Point();    // pt is a managed variable, subject to g.c.
fixed ( int* p = &pt.x )
{  // must use fixed to get address of pt.x and
   *p = 1;     //   pin pt in place while we use the pointer
}
```

You can initialize a pointer with the address of an array or a string:

```
fixed (int* p = arr) ...  // equivalent to p = &arr[0]
fixed (char* p = str) ... // equivalent to p = &str[0]
```

You can initialize multiple pointers, as long as they are all of the same type:

```
fixed (byte* ps = srcarray, pd = dstarray) {...} To initialize pointers
of different type, simply nest fixed statements:
     fixed (int* p1 = &p.x)
          fixed (double* p2 = &array[5])
      // do something with p1 and p2
```

Pointers initialized in fixed statements cannot be modified.
After a statement is executed, any pinned variables are unpinned and subject to garbage collection. Therefore, do not point to those variables outside the fixed statement.

In unsafe mode, you can allocate memory on the stack, where it is not subject to garbage collection and therefore does not need to be pinned.

Listing 19.4

```
using System;
class Point {
   public int x, y;
}
class FixedTest {
   // unsafe method: takes pointer to int
```

```
    unsafe static void SquarePtrParam (int* p)
    {
       *p *= *p;
    }
    public static void Main()
    {
       Point pt = new Point();
       pt.x = 5;
       pt.y = 6;
       unsafe
       {
          // pin pt in place:
          fixed (int* p = &pt.x)
          {
             SquarePtrParam (p);
          }
          // pt now unpinned
       }
       Console.WriteLine ("{0} {1}", pt.x, pt.y);
    }
}
```

19.4 Security

Most application and component developers should not need to do anything special in order to work with the .NET security system and benefit from the safety protection it provides.

The only code that represents an exception and requires more in- depth knowledge and special consideration of the security system is secure libraries. These are the codes that distinguish between the secure managed world and the unmanaged native code (that is outside the ability of the .NET security infrastructure to enforce). These libraries typically must be highly trusted to work, and are the areas in managed code where programming errors can potentially expose a security vulnerability. Code access security can't eliminate the potential for human error, but compared to the much larger volume of application code that uses a few secure libraries, the amount of code that requires close scrutiny is substantially reduced.

Security

.NET security protects code and data from misuse and damage by other code by enforcing security restrictions on managed code. .NET applications request the permissions they need and the security policy established by an administrator chooses to grant permissions based on determination of trust, or in some cases (if the request is too much) refuses to run the code. Trust is based on evidence about the code such as a digital signature, where the code comes from, and so forth. Depending on whether granted or not,security enforces control to the respective code.

Permissions

.NET security allows code to use protected resources only if it has "permission" to do so. To express this, .NET uses the concept of permissions, which represent the right for code to access protected resources. Code requests the permissions it needs, and the security policy applied by .NET determines which permissions the code is actually granted.

.NET provides code access permission classes.Each of these classes encapsulates the ability to access a particular resource. You use these permissions to indicate to .NET what your code needs to be allowed to do and to indicate what your code's callers must be authorized to do. Policy also uses these objects to determine what permissions to grant to code.

Policy Enforcement of security policy is what makes .NET managed code safe. Every assembly that loads is subject to security policy that grants code permissions based on trust, with trust based on evidence about the code. See the .NET documentation for information on administering security policy.

19.5 Modifying Security Permissions in C#

There are two ways to modify the current security permissions in C# by using calls to permission classes in the .NET frameworks, or by using security permission attributes.

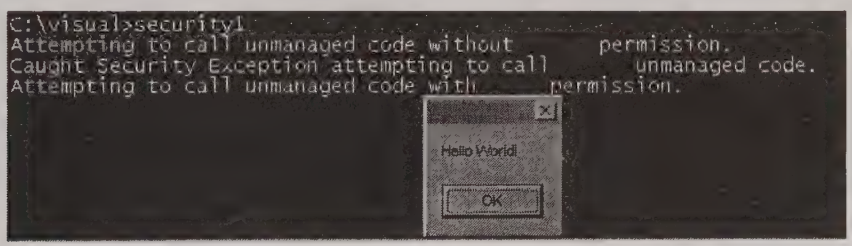
Here's an example of using .NET frameworks calls to remove the UnmanagedCode permission:

```
000: // Security\SecurityClasses.cs
001: using System;
002: using System.Security;
003: using System.Security.Permissions;
004: using System.Runtime.InteropServices;
005:
006: class NativeMethods
007: {
008:     //
009:      // this is a call to unmanaged code. Executing this method
requires
010:     // the UnmanagedCode security permission. Without this permis-
sion
011:     // an attempt to call this method will throw a SecurityException
012:      //
013:      [DllImport("user32.dll")]
014:     public static extern int MessageBox(uint hWnd, string lpText,
string lpCaption, uint uType);
015: }
016:
017: class MainClass
018: {
019:     private static void CallUnmanagedCodeWithoutPermission()
020:     {
021:         //
```

```
022:         // create a security permission object to describe the
UnmanagedCode permission
023:         //
024:         SecurityPermission perm = new
SecurityPermission(SecurityPermissionFlag.UnmanagedCode);
025:
026: //
027: // remove the UnmanagedCode from our current set of
       permissions
028:         //
029:         perm.Deny();
030:
031:         try
032:         {
033: Console.WriteLine("Attempting to call unmanaged
     code without permission.");
034: NativeMethods.MessageBox(0, "Hello World!", "", 0);
035: Console.WriteLine("Called unmanaged code without
      permission. Whoops!");
036:         }
037:         catch (SecurityException)
038:         {
039: Console.WriteLine("Caught Security Exception attempting to call
unmanaged code.");
040:       }
041:  }
042:
043:     private static void CallUnmanagedCodeWithPermission()
044:     {
045:         //
046: // create a security permission object to describe the UnmanagedCode
permission
047:         //
048:             SecurityPermission perm = new
SecurityPermission(SecurityPermissionFlag.UnmanagedCode);
049:
050:         //
051: //add the UnmanagedCode to our current set of permissions
052:         //
053:         perm.Assert();
054:
055:         try
056:         {
```

```
057: Console.WriteLine("Attempting to call unmanaged code with permis-
sion."); 058:          NativeMethods.MessageBox(0, "Hello World!", "",
0);
059: Console.WriteLine("Called unmanaged code with permission.");
060:         }
061:         catch (SecurityException)
062:         {
063:Console.WriteLine("Caught Security Exception attempting to call
unmanaged code. Whoops!");
064:         }
065:     }
066:
067:     public static void Main()
068:     {
069:         CallUnmanagedCodeWithoutPermission();
070:         CallUnmanagedCodeWithPermission();
071:     }
072: }
```

The output of this program shall get displayed as shown below:


C:\visual>security1
Attempting to call unmanaged code without permission.
Caught Security Exception attempting to call unmanaged code.
Attempting to call unmanaged code with permission.
Hello World!
OK

Here is the same example using attributes for the security permissions:

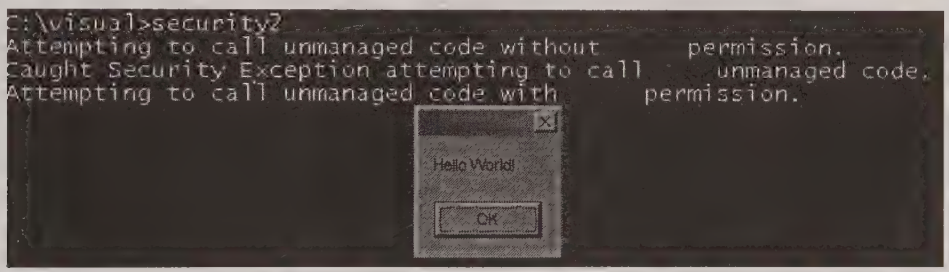
Listing 19.6

```
000: // Security\SecurityAttributes.cs
001: using System;
002: using System.Security;
003: using System.Security.Permissions;
004: using System.Runtime.InteropServices;
005:
006: class NativeMethods
007: {
008:     //
009:     // this is a call to unmanaged code. Executing this method
requires
010:     // the UnmanagedCode security permission. Without this per-
mission
011:     // an attempt to call this method will throw a SecurityException
```

```
012:     //
013:     [DllImport("user32.dll")]
014:     public static extern int MessageBox(uint hWnd, string lpText,
string                    lpCaption, uint uType);
015: }
016:
017: class MainClass
018: {
019:     //the security permission attached to this method will
020:     // remove the UnmanagedCode
021:     // permission from the current set of permissions for
022:     // the duration of the call to this method
023:[SecurityPermission(SecurityAction.Deny,Flags=SecurityPermissionFlag.
UnmanagedCode)]
024:     private static void CallUnmanagedCodeWithoutPermission()
025:     {
026:         try
027:         {
028:           Console.WriteLine(
        "Attempting to call unmanaged code without permission.");
029:           NativeMethods.MessageBox(0, "Hello World!", "", 0);
030:           Console.WriteLine(
        "Called unmanaged code without permission. Whoops!"); 031:
}
032:         catch (SecurityException)
033:         {
034:               Console.WriteLine(
        "Caught Security Exception attempting to call unmanaged code.");
035:         }
036:     }
037:
038:     // the security permission attached to this method will
039:     // add the UnmanagedCode permission
040:     // to the current set of permissions for the duration of
041:     // the call to this method
042:[SecurityPermission(SecurityAction.Assert,Flags=SecurityPermissionFlag.
UnmanagedCode)]
043:     private static void CallUnmanagedCodeWithPermission()
044:     {
045:         try
046:         {
047:           Console.WriteLine(
                  "Attempting to call unmanaged code with permis-
sion.");
048:             NativeMethods.MessageBox(0, "Hello World!", "",  0);
```

```
049:             Console.WriteLine("Called unmanaged code with permis-
sion.");
050:         }
051:         catch (SecurityException)
052:         {
053:             Console.WriteLine(
"Caught Security Exception attempting to call unmanaged code. Whoops!");
054:         }
055:     }
056:
057:     public static void Main()
058:     {
059:         CallUnmanagedCodeWithoutPermission();
060:         CallUnmanagedCodeWithPermission();
061:     }
062: }
```

The output of this program shall get displayed as shown below:


C:\visual>security2
Attempting to call unmanaged code without permission.
Caught Security Exception attempting to call unmanaged code.
Attempting to call unmanaged code with permission.
Hello World!
OK

19.6 Security and Performance

The .NET security system prevents malicious code downloaded from the network from damaging your computer system. However, these security checks are not without cost, even if your code never throws a SecurityException. You can disable the UnmanagedCode security demand when making calls to unmanaged code by using the SuppressUnmanagedCodeSecurity attribute (line 016 below).

Listing 19.7

```
000: // Security\SuppressSecurity.cs
001: using System;
002: using System.Security;
003: using System.Security.Permissions;
004: using System.Runtime.InteropServices;
005:
006: class NativeMethods
007: {
```

```
008: //
009: // this is a call to unmanaged code. Executing this method requires
010: // the UnmanagedCode security permission. Without this permission
011: // an attempt to call this method will throw a SecurityException
012: //
013: // NOTE:The SuppressUnmanagedCodeSecurity attribute disables the
check for 014: // NOTE: the UnmanagedCode permission. Be Careful!
015: //
016:     [SuppressUnmanagedCodeSecurity()]
017:     [DllImport("user32.dll")]
018:     internal static extern int MessageBox(uint hWnd, string lpText,
string                      lpCaption, uint uType);
019: }
020:
021: class MainClass
022: {
023:     // the security permission attached to this method will remove
the
024:     // UnmanagedCode permission from the current set of permis-
sions for the
025:     // duration of the call to this method
026:     //
027:[SecurityPermission(SecurityAction.Deny,Flags=Security
PermissionFlag.UnmanagedCode)]
028:     private static void CallUnmanagedCodeWithoutPermission()
029:     {
030:         try
031:         {
032:         // We've disabled the UnmanagedCode security check
033:         // on the call below. However, the unmanaged
034:         // call only displays UI. We'll ensure security
035:         // by only allowing the unmanaged call if
036:         // we have UI permission.
037:          UIPermission     uiPermission     =     new
UIPermission(PermissionState.Unrestricted);
038:         uiPermission.Demand();
039:
040:         Console.WriteLine(
        "Attempting to call unmanaged code without UnmanagedCode per-
mission.");
041:         NativeMethods.MessageBox(0, "Hello World!", "",  0);
042:         Console.WriteLine(
        "Called unmanaged code without UnmanagedCode permission.");
043:         }
```

```
044:          catch (SecurityException)
045:            {
046:                 Console.WriteLine(
              "Caught Security Exception attempting to call unmanaged
code.");
047:            }
048:      }
049:
050:      // the security permission attached to this method will
051:      // add the UnmanagedCode permission to the current set of
permissions
052:      // for the duration of the call to this method
053:      //
054:[SecurityPermission(SecurityAction.Assert,Flags=SecurityPermissionFlag.
UnmanagedCode)]
055:      private static void CallUnmanagedCodeWithPermission()
056:      {
057:          try
058:          {
059:  Console.WriteLine("Attempting to call unmanaged code with permis-
sion."); 060:          NativeMethods.MessageBox(0, "Hello World!", "",
0);
061:            Console.WriteLine("Called unmanaged code with permis-
sion.");
062:          }
063:          catch (SecurityException)
064:          {
065:            Console.WriteLine(
          "Caught Security Exception attempting to call unmanaged code.
Whoops!");
066:          }
067:      }
068:
069:      public static void Main()
070:      {
071:          CallUnmanagedCodeWithoutPermission();
072:          CallUnmanagedCodeWithPermission();
073:      }
074: }
```

Here it may be noted that, the aforesaid example allows both the unmanaged calls to succeed even though the first call doesn't have UnmanagedCode permission. When using the 'SuppressUnmanagedCodeSecurityAttribute', you should use other security checks to ensure that you don't incur a security hole. In the above example, this is done by adding the Demand for the UIPermission (line 038) before the unmanaged call, which ensures that the caller has permission to display UI.

20

Differences between C++ and C#

20.1 Simplicity

The most daunting feature of C++ is that one should remember when to use the -> pointer indicator, when to use the :: for a class member, and when to use the dot. C# recognizes thisdraw-back of the C++ programming language and simplifies it. In C#, everything is represented by a dot. Whether you are looking at members, classes, name-spaces, references, or whatever else, you do not need to track which operator to use.

Another hurdle with working in C and C++ is that one has to figure out exactly what type of data type to use. In C#, a Unicode character is no longer a wchar_t; it's a char. A 64-bit integer is a long, not an __int64. And a char is a char. There is no other char, unsigned char, signed char, and wchar_t to track. We will talk more about data types later in this article.

The third problem that you will come across in C and C++ is that integers are used as Booleans, causing assignment errors when you confuse = and ==. C# separates these two types, providing a separate bool type that solves this problem. A bool can be true or false, and cannot be converted into other types. Similarly, an integer or object reference cannot be tested to be true or false—it must be compared to zero (or to null in the case of the reference). If we write a code like the following in C++: int i; if (i) . . .

You need to convert that into something like this for C#: int i; if (i != 0) . . .

Another programmer-friendly feature is the improvement over C++ in the way switch statements work. In C++, we can write a switch statement that fell through from case to case. For example, in the following code:

```
 switch (i) {
case 1:
        FunctionA();
case 2:
        FunctionB();
        Break;
 }
```

would call both FunctionA and FunctionB if i was equal to 1. C# works like Visual Basic, putting an implied break before each case statement. If you really do want the case statement to fall through, you can rewrite the switch block like this in C#:

```
switch (i) {
case 1:
        FunctionA();
        goto case 2;
 case 2:
        FunctionB();
        Break;
}
```

20.2 Consistency

C# unifies the type system by letting you view every type in the language as an object. Whether you are using a class, a struct, an array, or a primitive, you will be able to treat it as an object. Objects are combined into namespaces, which allow you to access everything programmatically. This means that instead of putting 'includes' in your file like :

```
*  #include <stdlib.h> *  #include <stdio.h> *  #include <string.h>
```

you can include a particular namespace in your program to gain access to the classes and objects contained within it by employing:

```
using System;
```

20.3 Modernity with Safety

Just like coding languages, the needs of programmers evolve over time.C and C++ provide reliable transportation, but lack some of the features that programmers desire. This is one of the reasons why developers have taken recourse to the Java language over the past few years. C# goes back to the basics and emerges with several features that programmers wish for in C++. Garbage collection is one example—everything gets cleaned up when it is no longer referenced. However, garbage collection can have a price. It makes problems caused by certain risky behavior (using unsafe casts and stray pointers,for example) difficult to diagnose and potentially devastating to a program. To compensate for this, C# implements type safety to ensure application stability. C# has a richer intrinsic model for error handling than C++.There are dozens of unchecked HRESULTs in C++, and when a call fails, the program always ends up displaying an 'Error:There was an error' message. C# improves on this situation by providing integral support for throw, try—catch, and try—finally as language elements. True, you could do this as a macro in C++, but now it's available right out of the box.

Part of a modern language is the ability to actually use it for something. It seems simple enough, but many languages completely ignore the needs for financial and time-based data types. They are too old economy or something. Borrowing from languages like SQL, C# implements built-in support for data types like decimal and string, and lets you implement new primitive types that are as efficient as the existing ones.

You will also be glad to see that C# takes a more modern approach to debugging. The traditional way to write a debuggable program in C++ was to sprinkle it with #ifdefs and indicate that large sections of code would only be executed during the debugging process. You would end up with two implementations—a debug build and a retail build, with some of the calls in the retail build going to functions that do nothing. C# offers the conditional keyword to control program flow based on defined tokens. Finally, C# is designed to be easy to parse, so vendors can create tools that allow source browsing and two-way code generation.

20.4 Object Oriented

C++ is object oriented.But working with multiple inheritance in C++ can be very tedious. That is why C# swaps multiple inheritance in favor of native support for the COM+ virtual object system. Encapsulation, polymorphism, and inheritance are preserved and besides,rendered simpler. C# discards the entire concept of global functions, variables, and constants. Instead, you can create static class members, makings C# code easier to read and less prone to naming conflicts.

Speaking of naming conflicts, by default, C# methods are nonvirtual, requiring an explicit virtual modifier. It is far harder to accidentally override a method,than to provide correct versioning. Class members in C# can be defined as private, protected, public, or internal. You retain full control over their encapsulation. Methods and operators can be overloaded in C#, using a syntax that's a lot easier than the one used by C++. However, you can't overload global operator functions—the overloading is strictly local in scope. The overloading of method F below is an example of what this looks like:

```
interface ITest {
    void F();              // F()
    void F(int x);         // F(int)
    void F(ref int x);     // F(ref int)
    void F(out int x);     // F(out int)
    void F(int x, int y); // F(int, int)
    int F(string s);       // F(string)
    int F(int x);          // F(int)
}
```

The COM+ component model is supported through the implementation of delegates—the object-oriented equivalent of function pointers in C++. Interfaces support multiple inheritance. Classes can privately implement internal interfaces through explicit member implementations, without the consumer ever knowing about it.

20.5 Type Safety

Type safety promotes robust programs. Several features that promote proper code execution (and more robust programs) in Visual Basic have been included in C#. For example, all dynamically allocated objects and arrays are initialized to zero. Although C# doesn't automatically initialize local variables, the compiler will warn you if you use one before you initialize it. When you access an array, it is automatically range checked. Unlike C and C++, you can't overwrite unallocated memory.

In C# you can't create an invalid reference. All casts are required to be safe, and you cannot cast between integer and reference types. Garbage collection in C# ensures that you don't leave references dangling around your code. Hand-in-hand with this feature is overflow checking. Arithmetic operations and conversions are not allowed if they overflow the target variable or object. Of course, there are some valid reasons to want a variable to overflow. If you do, you can explicitly disable the checking. As we have discussed earlier, the data types supported in C# are somewhat different from what you might be used to in C++. For instance, the char type is 16 bits. Certain useful types, like decimal and string, are built in.

20.6 Scalability

C and C++ require all sorts of often-incompatible header files before you can compile even a simple code. C# gets rid of these headers by combining the declaration and definition of types. It also directly imports and emits COM+ metadata, making incremental compiles much easier. When a project gets large enough, you might want to split up your code into smaller source files. C# doesn't have any restrictions about where your source files live or what they are named. When you compile a C# project, you can think of it as concatenating all the source files, then compiling them into one big file. You don't have to track which headers go where, or which routines belong in which source file. This also means that you can move, rename, split, or merge source files without breaking the compile.

20.7 Version Support

DLL Hell is a constant problem for users and programmers alike. There is nothing a programming language can do to keep a library author from messing around with a published API. However, C# was designed to make versioning far easier by retaining binary compatibility with existing derived classes. When you introduce a new member in a base class as one that exists in a derived class, it doesn't cause an error. However, the designer of the class must indicate whether the method is meant as an override or as a new method that just hides the similar inherited method.

As we have already observed earlier, C# works with a namespace model. Classes and interfaces in class libraries must be defined in hierarchical namespaces instead of in a flat model. Applications can explicitly import a single member of a namespace, so there won't be any collisions when multiple namespaces contain similarly named members. When you declare a namespace, subsequent declarations are considered to be part of the same declaration space. Therefore, if your code looks like this;

Listing 20.1

```
namespace MSDNMag.Article {  class Author
    {
        ...
    }
} namespace MSDNMag.Article {
    class Topic
    { ...
    }
}
```

you can express the same code like:

```
namespace MSDNMag.Article {
    class Author
    {
    ...
    }
    class Topic
    {
        ...
    }
}
```

20.8 Compatibility

Four types of APIs are common on the Windows platform and C# supports all of them. The old-style C APIs have integrated support in C#. Applications can use the N/Direct features of COM+ to call C-style APIs. C# provides transparent access to standard COM and OLE Automation APIs and supports all data types through the COM+ runtime. Most importantly, C# supports the COM+ Common Language Subset specification. If you have exported any entities that aren't accessible from another language, the compiler can optionally flag the code. For instance, a class can't have two members—runJob and runjob—because a case-insensitive language would choke on the definitions. When you call a DLL export, you need to declare the method, attach a sysimport attribute, and specify any custom marshaling and return value information that overrides the COM+ defaults. The following shows how to write a Hello World program that displays its message of cheer in a standard Windows message box.

Listing 20.2

```
class HelloWorld {
    [sysimport(dll = "user32.dll")]
    public static extern int MessageBoxA(int h, string m,
                                         string c, int type);

    public static int Main()
    {
        return MessageBoxA(0, "Hello World!", "Caption", 0);
    }
}
```

Each COM+ type maps to a default native data type, which COM+ uses to marshal values across a native API call. The C# string value maps to the LPSTR type by default, but it can be overridden with marshaling statements (Listing 20.1).

Listing 20.3

```
using System; using System.Interop;
class HelloWorld {
    [dllimport("user32.dll")]
    public static extern int MessageBoxW(
```

```
            int h,
            [marshal(UnmanagedType.LPWStr)] string m,
            [marshal(UnmanagedType.LPWStr)] string c,
            int type);

        public static int Main()
        {
            return MessageBoxW(0, "Hello World!", "Caption", 0);
        }
    }
```

In addition to working with DLL exports, you can work with classic COM objects in several ways: create them with CoCreateInstance, query them for interfaces, and call methods on them. If you want to import a COM class definition for use within your program, you must take two steps. First, you must create a class and use the comimport attribute to mark it as related to a specific GUID. The class you create can't have any base classes or interface lists, nor can it have any members.

```
// declare FilgraphManager as a COM classic
coclass [comimport, guid("E436EBB3-524F-11CE-9F53-0020AF0BA770")]
class FilgraphManager .{ }
```

After the class is declared in your program, you can create a new instance of it with the new keyword (which is equivalent to the CoCreateInstance function).

```
class MainClass {
    public static void Main()
    {
        FilgraphManager f = new FilgraphManager();
    }
}
```

You can query interfaces indirectly in C# by attempting to cast an object to a new interface. If the cast fails, it will throw a System.InvalidCastException. If it works, you will have an object that represents that interface.

```
FilgraphManager graphManager = new FilgraphManager();
IMediaControl mc = (IMediaControl) graphManager;
mc.Run(); // If the cast succeeds, this line will work.
```

20.9 Flexibility

It is definitely true that C# and COM+ create a managed, type-safe environment. However, it is equally true that some real-world applications need to get to the native code level—either for performance considerations or to use old-style, unmodernized APIs from other programs. We have discussed the ways to use APIs and COM components from the C# program. C# lets you declare unsafe classes and methods that contain pointers, structs, and static arrays. These

methods won't be type-safe, but they will execute within the managed space; and so, we do not need to marshal boundaries between safe and unsafe code. These unsafe features are integrated with the COM+ EE and code access security in COM+. This means that a developer can pin an object so that the garbage collector will pass over them when it's doing its work. Unsafe code won't be executed outside a fully trusted environment. Programmers can even turn-off garbage collection while an unsafe method is executing.

20.10 Visual Basic.NET

To rapidly build enterprise Web applications, developers must rely on business logic that is scalable, robust, and reusable. Over the past several years, object-oriented programming has emerged as the primary methodology for building systems that meet these requirements. Using object-oriented programming languages helps make large-scale systems easier to understand, simpler to debug, and faster to update.

To enable Visual Basic developers to benefit from object-oriented design and to simplify the development of enterprise Web applications, full object-oriented language features, including implementation inheritance, will be supported in the next version of Visual Basic—Visual Basic.NET. With these new language features, Visual Basic.NET will deliver all the power required to quickly and effectively develop enterprise-critical applications while maintaining the instant accessibility that has made it the world's most popular development tool.

Visual Basic.NET will provide a first class object-oriented programming language with new features such as implementation inheritance, overloading, and parameterized constructors. Additionally, developers will be able to create highly scalable code with explicit free threading and highly maintainable code with the addition of modernized language constructs like structured exception handling. Visual Basic will provide all the language characteristics that developers need to create robust, scalable distributed Web applications with the following new features:

New object oriented programming features

- Inheritance
- Overloading
- Parameterized Constructors

Additional modernized language features

- Free Threading
- Structured Exception Handling
- Strict Type Checking
- Shared Members
- Initializers

20.11 A History of Language Innovation

The Visual Basic language has a long history of updates that map to fundamental changes in the Windows platform. For example, the significant changes made to QuickBasic to support Windows 3.0 GUI development resulted in the first release of Visual Basic. In Visual Basic 4.0, the shift to COM-based programming resulted in language constructs for creating DLLs. And in Visual Basic 5.0, the language evolved to support the creation of COM controls.

With each successive revision, the popularity of Visual Basic has soared. The power that the new Visual Basic object-oriented language features provide developers building enterprise Web applications will most certainly continue this trend.

Developers can use the new keyword Inherits to derive from an existing class.

```
Class1
        Function GetCustomer()
        ...
        End Function
Class2
        Inherits Class1
        Function GetOrders()
        ...
        End Function
```

The Inherits statement supports all the usual properties associated with inheritance. Instances of the derived class support all methods and interfaces supported by the base class. And of course, the derived class can extend the set of methods and interfaces supported by the base class.

The derived class can override methods defined in the base class using the Overrides keyword. In order to reduce programming errors, Visual Basic prevents you from accidentally overriding a function; only functions that are declared "Overridable" are allowed to be overridden in derived classes.

20.12 Overloading

Visual Basic now allows function overloading, which gives developers the ability to create different versions of a Sub or Function that have the same name, but different argument types.

Overloading is especially useful when your object model dictates that you employ similar names for procedures that operate on different data types. For example, a class that can display several different data types could have Display procedures that look like this:

Overloads Sub Display (theChar As Char) ... Overloads Sub Display (theInteger As Integer) ... Overloads Sub Display (theDouble As Double)

Without overloading, you will need to create distinct names for each procedure or use a Variant parameter. Overloading provides a more explicit and efficient way to handle multiple data types.

20.13 Parameterized Constructors

Parameterized constructors (or simply "constructors") allow you to create a new instance of a class while simultaneously passing arguments to the new instance. Constructors are essential for object oriented programming since they allow user-defined construction code to be passed parameters by the creator of the instance. They simplify client code by allowing a new object instance to be created and initialized in a single expression.

Additional Modernized Language Features Visual Basic.NET adds a number of additional constructs that simplify the development of more robust, scalable applications. These features include free threading, structured exception handling, strict type safety as well as productivity features like initializers, and shared members.

20.14 Free Threading

Today when developers create applications in Visual Basic, the code that they write is synchronous. That means that each line of code must be executed before the next one. When developing Web applications, scalability is key. Developers need tools that enable concurrent processing.

With the inclusion of free threading, developers can spawn a thread which can perform some long-running task, execute a complex query, or run a multipart calculation while the rest of the application continues, providing asynchronous processing.

Listing 20.4

```
Sub CreateMyThread()
      Dim b As BackGroundWork
      Dim t As Thread
      Set b = New BackGroundWork()
      Set t = New Thread(New ThreadStart(AddressOf      b.Doit))
      t.Start
End Sub Class BackGroundWork
      Sub DoIt()

      _
      End Sub
End Class
```

20.15 Structured Exception Handling

Developing enterprise applications requires the construction of reusable, maintainable components. One challenging aspect of the Basic language in past versions of Visual Basic was its support for error handling. Developers have found that a consistent error-handling scheme means a great deal of duplicated code. Error handling using the existing On Error GoTo statement sometimes slows the development and maintenance of large applications. Its very name reflects some of these problems: As the GoTo implies, when an error occurs, control is transferred to a labeled location inside the subroutine. Once the error code runs it must often be diverted via another cleanup location via a standard GoTo, which finally uses yet another GoTo or an Exit out of the procedure. Handling several different errors with various combinations of Resume and Next quickly produces illegible code and it leads to frequent bugs when execution paths aren't completely thought out.

With Try...Catch...finally, these problems go away, developers can nest their exception handling, and there is a control structure for writing cleanup code that executes in both normal and exception conditions.

```
Sub SEH()
     Try
          Open "TESTFILE" For Output As #1
          Write #1, CustomerInformation
     Catch
          Kill "TESTFILE"
     Finally
          Close #1
     End try
End Sub
```

20.16 Strict Type Checking

Today the Visual Basic language is very liberal in the implicit type coercions that it will generate. For assignment and for parameter passing other than by reference, the Visual Basic compiler will allow nearly any data type to be converted to any other type by generating runtime coercion. The runtime coercion operation will fail if the value that is to be converted cannot be converted without data loss. Through the addition of a new compilation option, Visual Basic can generate compile-time errors for any conversions that may cause an error at runtime. Option Strict improves type safety by generating errors when a conversion is required which could fail at runtime or which, like the automatic conversion between numeric types and strings, is unexpected by the user.

20.17 Shared Members

Shared members are data and function members of classes that are shared by all instances of the class. Sharing a single instance of a data member or function among all instances of a class is required in a Visual Basic application with inheritance. A shared data member exists independently of any particular instance of the class. A *shared method* is a method, which unlike normal methods, is not implicitly passed an instance of the class. For this reason no unqualified references to non-shared data members is allowed in a shared method. Public shared members can be accessed remotely and they can be late-bound from an instance of the class.

20.18 Initializers

Visual Basic.NET supports initialization of variables on the line in which they are declared. *Initializers* can be used anywhere including inside a control structure. The semantics of a procedure level declaration, which includes an initializer, is the same as a declaration statement immediately followed by an assignment statement. In other words, this statement:

```
Dim X As Integer = 1
```

is equivalent to the statement:

```
Dim X As Integer X = 1
```

20.19 Conclusion

Visual Basic is now a first class object-oriented programming language. Using Visual Basic.NET, developers will be able to create highly scalable code with explicit Free Threading. The code they write will be highly maintainable with the addition of modernized language constructs like Structured Exception Handling. Visual Basic will provide all the language characteristics that developers need to create robust, scalable distributed Web applications.

20.20 A Comparative Overview of C#

This section will focus on the new ways of programming that C# offers, and how it intends to improve upon its two closest neighbors, Java and C++. C# improves on C++ in a similar way to Java in many respects. Consequently, we are not going to go into discussing things like the benefits of a single-rooted object hierarchy. Here we begin with a brief summary of the similarities between C# and Java, and then go into exploring the new C# features.

20.21 Background

In June 2000, Microsoft announced both the .NET platform and a new programming language called C#. C# is a strongly-typed object-oriented language designed to give the optimum blend of simplicity, expressiveness, and performance. The .NET platform is centered around a Common Language Runtime(similar to a JVM) and a set of libraries which can be exploited by a wide variety of languages which are able to work together by all compiling to an intermediate language (IL). C# and .NET are a little symbiotic: some features of C# are there to work well with .NET, and some features of .NET are there to work well with C# (though .NET aims to work well with many languages).

The C# language was built with the hindsight of many languages, but most notably Java and C++. It was co-authored by Anders Hejlsberg (who is famous for the design of the Delphi language), and Scott Wiltamuth.

20.21.1 C# and Java

Here we shall go through the features that C# and Java share—these are intended to improve on C++. These features are not the focus of this section, but it is very important for us to be aware of the similarities.

- Compiles into machine-independent language-independent code which runs in a managed execution environment.
- Garbage Collection coupled with the elimination of pointers (in C# restricted use is permitted within code marked unsafe).
- Powerful reflection capabilities No header files, all code scoped to packages or assemblies, no problems declaring one class before another with circular dependencies.
- Classes all descend from object and must be allocated on the heap with new keyword.
- Thread support by putting a lock on objects when entering code marked as locked/ synchronized

- Interfaces, with multiple-inheritance of interfaces, single inheritance of implementations
- Inner classes
- No concept of inheriting a class with a specified access level
- No global functions or constants, everything belongs to a class
- Arrays and strings with lengths built-in and bounds checking
- The "." operator is always used, no more ->,
- :: operators
- null and boolean/bool are keywords
- All values are initialized before use
- Can't use integers to govern if statements
- Try Blocks can have a finally clause

20.21.2 Properties

Properties will be a familiar concept to Delphi and Visual Basic users. The motivation is for the language to formalize the concept of getter/setter methods, which is an extensively used pattern, particularly in RAD (Rapid Application Development) tools.

This is a typical code that you might write in Java or C++:

```
foo.setSize (foo.getSize () + 1); label.getFont().setBold (true);
```

The same code can be written in C# as:

```
foo.size++; label.font.bold = true;
```

The C# code is immediately more readable by those who are using foo and label. There is similar simplicity when implementing properties:

```
Java/C++:
public int getSize() {
     return size;
}
public void setSize (int value) {
     size = value;
}
C#:
public int Size {
     get {return size;
     }
     set {size = value;
     }
}
```

In particularly for read/write properties, C# provides a cleaner way of handling this concept. The relationship between a get and set method is inherent in C#, while they have to be maintained in Java or C++. There are many benefits to this approach. It encourages programmers to think in terms of properties, and whether that property is most natural as read/write vs read only, or

whether it really should not be a property at all. If you wish to change the name of your property, you only have one place to look (there are instances in which getters and setter are placed several hundred lines away from each other). Comments only have to be made once, and won't get out of sync with each other. It is feasible that an IDE could help out here (and they do), but one should remember an essential principle in programming is to try to make abstractions which model our problem space well. A language which supports properties will reap the benefits of that better abstraction.

One possible argument against this being a benefit is that you do not know if you're manipulating a field or a property with this syntax. However, almost all classes with any real complexity designed in Java (and certainly in C#) do not have public fields. Fields typically have a reduced access level (private/protected/default) and are only exposed through getter/setters, which means one may as well have the nicer syntax. It is also totally feasible an IDE could parse the code, highlighting properties with a different colour, or provide code completion information indicating if it is a property or not. It should also be noted that if a class is designed well, then a user of that class should only worry about the specification of that class, and not its implementation. Another possible argument is that it is less efficient. As a matter a fact, good compilers can in-line the default getter which merely returns a field, making it just as fast as field. Finally, even if using a field is more efficient that a getter/setter, it is a good thing to be able to change the field to a property later without breaking the source code which relies on the property.

20.21.3 Indexers

C# provides indexers which allow objects to be treated like arrays, except that like properties, each element is exposed with a get and/or set method.

Listing 20.5

```
public class Skyscraper {
    Story[] stories;
    public Story this [int index] {
        get {
            return stories [index];
        }
        set {
            if (value != null) {
                stories [index] = value;
            }
        }
    }
    ...
}
Skyscraper empireState = new Skyscraper (...); empireState [102] = new
Story ("The Top One", ...);
```

20.21.4 Delegates

A delegate can be thought of as a type-safe object-oriented function pointer, which is able to hold multiple methods rather than just one. Delegates handle problems which would be solved with function pointers in C++, and interfaces in Java. It improves on the function pointer approach by being type safe and being able to hold multiple methods. It improves on the interface approach by allowing the invocation of a method without the need for inner-class adapters or extra code to handle multiple-method invocations. The most important use of delegates is for event handling, which is in the next section (which gives an example of how delegates are used).

20.21.5 Events

C# provides direct support for events. Although event-handling has been a fundamental part of programming since programming began, there has been surprisingly little effort made by most languages to formalize this concept. If you look at how today's mainstream frameworks handle events, we've got examples like Delphi's function pointers (called *closures*), Java's inner class adaptors, and of course, the Windows API's message system. C# uses delegates along with the event keyword to provide a very clean solution to event handling. The best way to illustrate this is to understand an example showing the whole process of declaring, firing, and handling an event:

Listing 20.6

```
// The Delegate declaration which defines the signature
// of methods which can be invoked
public delegate void ScoreChangeEventHandler (int newScore, ref  bool
cancel);
// Class which makes the event
public class Game {
    // Note the use of the event keyword
    public event ScoreChangeEventHandler ScoreChange;
    int score;
     // Score Property
    public int Score {
         get {
            return score;
         }
         set {
            if (score != value) {
                bool cancel = false;
                ScoreChange (value, ref cancel);
                if (! cancel)
                    score = value;
            }
        }
    }
}
   // Class which handles the event
```

```
    public class Referee {
     public Referee (Game game) {
         // Monitor when a score changes in the game
                                    game.ScoreChange    +=    new
ScoreChangeEventHandler(game_ScoreChange);
     }
     // Notice how this method signature
    //matches the  ScoreChangeEventHandler's signature
     private void game_ScoreChange (int newScore, ref bool cancel)
    {
         if (newScore < 100)
             System.Console.WriteLine ("Good Score");
         else {
             cancel = true;
             System.Console.WriteLine ("No Score can be that high!");
         }
     }
}
// Class to test it all
public class GameTest {
     public static void Main () {
         Game game = new Game ();
         Referee referee = new Referee (game);
         game.Score = 70;
         game.Score = 110;
     }
}
```

In the GameTest we make a game, make a referee who monitors the game, then change the game's score to see what the referee does in response to this. With this system, the Game has no knowledge of the Referee at all, and simply lets any class listen and act on changes made to its score. The event keyword hides all the delegate's methods apart from += and -= to classes other than the class it is declared in. These operators allow you to add (and remove) as many event handlers as you want to the event.

You will probably first encounter this system in GUI frameworks, where the Game would be analogous to UI widgets which fire events according to user input, and the referee would be analogous to a form, which would process the events.

20.21.6 Enums

In case you do not know C, Enums let you specify a group of objects, e.g.:

Declaration:

```
public enum Direction {North, East, West, South};
```

Usage:

```
Direction wall = Direction.North;
```

It is a nice construct, and so the question perhaps is not why C# decides to have them, but rather, why did Java choose to omit them? In Java, you would have to go as follows:

Declaration:

```
public class Direction {
     public final static int NORTH = 1;
     public final static int EAST = 2;
     public final static int WEST = 3;
     public final static int SOUTH = 4;
}
```

Usage:

```
int wall = Direction.NORTH;
```

Despite the known fact that the Java version seems to express more, in reality it doesn't, and is less type-safe—by allowing you to accidentally assign wall to any int value without the compiler complaining. To be fair, in Java programming, we haven't wasted much time on writing a few extra tokens and tracking down an error because of lack of type-safety here, but nonetheless, this is nice to have. One benefit of C# is the nice surprise you get when debugging—if you put a break point on an enumeration which holds combined enumerations, it will automatically decode the direction to give you a human readable output, rather than a number you have to decode:

Declaration:

```
public enum Direction {North=1, East=2, West=4, South=8}; Usage: Direction
direction = Direction.North | Direction.West; if ((direction &
Direction.North) != 0)
     ....
```

If you put a breakpoint on the if statement, you will get the human readable version for direction rather than the number 5.

The mostly likely reason why enums are absent in Java is that you can get them by using classes instead. As discussed in previous sections, with classes alone, we are not able to express a feature in the world as well as we could with another construct. What are the benefits of the Java philosophy of 'if it can be done by a class, then don't introduce a new construct'? It would seem simplicity would be the biggest benefit—a shorter learning curve and the prevention of programmers having to think of multiple ways of doing things. Indeed the Java language has improved on C++ in many ways by aiming for simplicity, such as the elimination of pointers, the elimination of header files, and a single-rooted object hierarchy. However, a common aspect of all these simplifications is they actually make coding simpler. Leaving out constructs, we have looked at enums, properties and events so far—makes your coding more complicated.

20.21.7 Collections and the Foreach Statement

C# provides a shorthand for for-loops, which also encourages increased consistency for collections classes:

In Java or C++:

```
1. while (! collection.isEmpty()) {
```

```
        Object o = collection.get();
        collection.next();
    ...
2. for (int i = 0; i < array.length; i++)...
In C#:
```

1. foreach (object o in collection)... 2. foreach (int i in array)...
The C# for-loop will work on collection objects (arrays implement a collection). Collection objects have a GetEnumerator() method which returns an Enumerator object. An Enumerator object has a MoveNext method and a Current property.

20.21.8 Structs

It is helpful to view C# structs as a construct which makes the C# type system work elegantly, rather than just "a way to write really efficient code if you need to".

In C++, both structs and classes (objects) can be allocated on the stack/in-line or on the heap. In C#, structs are always created on the stack/in-line, and classes (objects) are always created on the heap. Structs indeed allow more efficient code to be created:

```
public struct Vector {
    public float direction;
    public int magnitude;
}
Vector[] vectors = new Vector [1000];
```

This will allocate the space for all 1000 vectors in one lump, which is much more efficient that had we declared Vector as a class and used a for-loop to instantiate 1000 individual vectors. What we have effectively done is declare the array like we would declare an int array in C# or Java:

```
int[] ints = new ints [1000];
```

C# simply allows you to extend the primitive set of types built in to the language. In reality, C# implements all the primitive types as structs. The int type merely aliases System.Int32 struct, the long type aliases System.Int64 struct etc. These primitive types are of course able to be treated specially by the compiler, but the language itself does not make such a distinction. The next section shows how C# takes advantage of this.

20.21.9 Type Unification

Most languages have primitive types (int, long, etc), and higher level types which are ultimately composed of primitive types. It is often useful to be able to treat primitive types and higher level types in the same way. For instance, it is useful to have collections which can take both ints as well as strings. Smalltalk achieves this by sacrificing some efficiency and treating ints and longs as types like String or Form. Java tries to avoid sacrificing this efficiency, and treats primitive types like in C or C++, but provides corresponding wrapper classes for each primitive—int is wrapped by Integer, double is wrapped by Double. C++'s templates allow code to be written which takes any type, so long as the operations done on that type are provided by that type.

C# provides a different solution to this problem. In the previous section, we have studied C# structs—and explained how primitive types like int were merely aliases for structs. This allows code like the following to be written, since structs have all the methods that the object class does:

```
int i = 5; System.Console.WriteLine (i.ToString());
```

If we want to use a struct as an object, C# will box the struct in an object for you, and unbox the struct when you need it again:

```
Stack stack = new Stack ();
stack.Push (i); // box the int
int j = (int) stack.Pop(); // unbox the int
```

Apart from a type-cast required when unboxing structs, this is a seamless way to handle the relationship between structs and classes. You should bear in mind that boxing does entail the creation of a wrapper object, though the CLR may provide additional optimization for boxed objects.

The designers of C# must have considered templates during their design process. Two main reasons for not using templates are suspected here. The first is messiness—templates can be difficult to mesh with object oriented features since they open up too many (confusing) design possibilities for programmers, and are difficult to work with reflection. The second is that templates wouldn't be very useful unless the .NET libraries like the collection classes used them. However, if the .NET classes used them, then the 20+ languages which use the .NET classes would have to work with templates too, which may be technically very difficult to achieve.

It is interesting to note that templates (generics) are being considered for inclusion in the Java language by the Java Community Process. It may be understood that each company starts singing each other's tunes when Sun says that '.NET suffers from lowest common denominator syndrome', and Microsoft says 'Java doesn't have cross-language support'.

20.21.10 Operator Overloading

Operator overloading allows programmers to build types which feel as natural to use as simple types (int, long, etc.). C# implements a stricter version of operator overloading in C++, but it allows classes such as the quintessential example of operator—overloading, the complex number class, to work well.

In C#, the == operator is a non-virtual (operators can't be virtual) method of the object class which compares by reference. When you build a class, you may define your own == operator. If you are using your class with collections, then you should implement the IComparable interface. This interface has one method to implement, called the CompareTo (object) method, which should return positive, negative, or 0 if "this" is greater, less than, or the same value as the object. You may choose to define <, <=, >=, > methods if you want users of your class to have a nicer syntax. The numeric types (int, long, etc) implement the IComparable interface.

Here is a simple example of how to deal with equality and comparisons:

Listing 20.7

```
public class Score : IComparable {
    int value;
    public Score (int score) {
        value = score;
    }
    public static bool operator == (Score x, Score y) {
        return x.value == y.value;
    }
    public static bool operator != (Score x, Score y) {
        return x.value != y.value;
    }
    public int CompareTo (object o) {
        return value - ((Score)o).value;
    }
}
```

Score a = new Score (5); Score b = new Score (5); Object c = a; Object d = b;

To compare a and b by reference:

```
System.Console.WriteLine((object)a=(object)b;
//false To compare a and b by value:
System.Console.WriteLine (a == b);
// true To compare c and d by reference:
System.Console.WriteLine (c == d);
// false To compare c and d by value:
System.Console.WriteLine (((IComparable)c).CompareTo (d) == 0);
true You could also add the <, <=, >=, > operators to the score class.
C# ensures at compile time that operators which are logically
paired (!= and ==, > and <, >= and <=), must both be defined.
```

20.21.10 Polymorphism

Virtual methods allow object oriented languages to express polymorphism. This means that a derived class can write a method with the same signature as a method in its base class, and the base class will call the derived class's method. By default in Java, all methods are virtual. In C#, like C++, the virtual keyword must be used so that the method will be called by the base class.

In C#, it is also required that the override keyword is needed to specify that a method should override a method (or implement an abstract method) of its base class.

```
Class B {
    public virtual void foo () {}
}
Class D : B {
    public override void foo () {}
}
```

Attempting to override a non-virtual method will result in a compile-time error unless the "new" keyword is added to the declaration, indicating the method is intentionally hiding the base class's method.

```
Class N : D {
    public new void foo () {}
}
N n = new N ();
n.foo(); // calls N's foo
((D)n).foo();
// calls D's foo
((B)n).foo();
// calls D's foo
```

In contrast to both C++ and Java, requiring the override keyword makes it more clear as to what methods are overridden when looking at source code. However, requiring the use of the virtual method has its pros and cons. The first pro is the slightly increased execution speed from avoiding virtual methods. The second pro is to make clear what methods are intended to be overridden. However, this pro can also be a con. Compare the default option of leaving out a final modifier in Java vs leaving out a virtual modifier in C++. The default option in Java may make your program slightly less efficient, but in C++ it may prevent extendibility, albeit unforeseen, by the implementer of the base class.

20.21.11 Interfaces

Interfaces in C# are similar to Java interfaces, but can be used with greater flexibility. A class may optionally "explicitly" implement an interface:

```
public interface ITeller {
    void Next ();
}
public interface IIterator {
    void Next ();
}
public class Clark : ITeller, IIterator {
    void ITeller.Next () {
    }
    void IIterator.Next () {
    }
}
```

This gives a class two benefits. First, a class can implement several interfaces without having to worry about naming conflicts. Second, it allows a class to 'hide' a method if it is not useful for general users of the class. Explicitly implemented methods are invoked by type-casting a class to the interface required:

```
Clark clark = new Clark ();
((ITeller)clark).Next();
```

20.21.12 Versioning

Solving versioning issues has been a major consideration in the .NET framework. Most of these considerations apply to assemblies. There are some quite impressive capabilities such as running multiple versions of the same assembly in the same process.

The C# language prevents software failures when new versions of code (most notably the .NET libraries) are made. The C# Language Reference explains this in detail, but I have summarized the problem with a highly condensed example:

1. In Java, supposing we deploy a class called D which derives a class distributed with the VM, called B. Class D has a method called foo which B does not have at the time of distribution. Later, an update is made to class B, which now includes a foo method, and the new VM is installed on the computer running software which uses class D. The software using class D may malfunction because the new implementation of class B will make a virtual call to class D, performing an action with an implementation totally unintended by class B. In C#, class D's foo would have been declared without the override modifier(which expresses the programmer's intention), so the runtime knows to make Class D's foo hide Class B's foo, rather than override it.
2. An interesting quote from the C# reference manual was: "C# addresses this versioning problem by requiring developers to clearly state their intent". Although the use of the word override is one way to state intention, it could also be automatically generated by the compiler by checking to see if a method performs (rather than declares) an override at the time of compilation. This means you can still have Java-like languages (which don't use virtual and override keywords) and still cope with versioning correctly.

20.21.13 Parameter Modifiers

'ref' parameter modifier

C# (as opposed to Java) lets you pass parameters by reference. The most obvious example to illustrate the point of this is the general purpose swap method. Unlike in C++, you must specify when calling as well as when declaring a method which takes ref parameters:

Listing 20.8

```
public class Test {
    public static void Main ()
    {
        int a = 1;
        int b = 2;
        swap (ref a, ref b);
    }
    public static void swap (ref int a, ref int b) {
       int temp = a;
       a = b;
       b = temp;
    }
}
```

'out' parameter modifier

There is also an 'out' keyword, which is a natural complement to the ref parameter modifier. While the ref modifier requires that a value is definitely assigned before passed into a method, the out modifier requires that the method's implementation definitely assigns the parameter a value before returning.

'params' parameter modifier

The params modifier may be added to the last parameter of a method so that the method accepts any number of parameters of a particular type. For example:

Listing 20.9

```
public class Test {
    public static void Main () {
        Console.WriteLine (add (1, 2, 3, 4).ToString());
    }
    public static int add (params int[] array) {
        int sum = 0;
        foreach (int i in array)
            sum += i;
        return sum;
    }
}
```

One of the most surprising things when you are learning Java is not being able to pass by reference. It turns out though that after a little while you seldom miss this functionality, and write code which doesn't use it. When I went over the C# specification for the first time, I often thought "Why have they got this or that functionality, I can code without it". With a little introspection, I realized this wasn't really an argument to show some functionality wasn't useful, but was more an argument to show that you get conditioned into getting by without it.
Java did a good thing when it simplified how parameters could be passed when you consider what C++ does. In C++, the parameters of a method and a method call pass in values or references, which in addition to having pointers, can lead to unnecessarily complicated code. C# makes passing by reference explicit for both the method and the method call, which greatly alleviates any confusion, thus achieving the same goal as Java—but with greater expressiveness. It is clearly a theme of C# to not hedge programmers into situations where they require a round-about way of achieving something. Remember those Java tutorials that suggest that to overcome the pass-by-reference problem, you should pass a 1-element array to holding your value, or make another class to hold that value?

20.21.14 Attributes

Both C# and Java contain information such as the access level of a field in the compiled code. C# generalizes this ability, so you can compile custom information about just about any element of code such as a class, method, field and even individual parameters. This information can be retrieved at run-time. Here is a very simple example of a class which makes use of attributes:

```
[AuthorAttribute ("Ben Albahari")] class A {
    [Localizable(true)]
    public String Text {
        get {return text;
        }
        ...
    }
}
```

Java uses a combination of /** */ and @tag comments to include additional information about classes and methods, but this information (apart from the @deprecated) is not built into the byte code. C# uses the pre-defined attributes ObsoleteAttribute so that the compiler can warn you against obsolete code (like @deprecated), and the ConditionalAttribute to enable conditional compilation. Microsoft's new XML libraries utilize attributes to express how fields should be serialized into XML, which means you can easily turn a class into XML and then reconstruct it again. Another apt use for attributes is for the creation of a really powerful class browsing tool. The C# Language Reference explains exactly how to create and use attributes.

20.21.15 Selection Statements

C# lets you govern a switch statement with an integral type, char, enum or (unlike C++ or Java) a string. In Java and C++ if you leave out a break after each case statement you risk having another case statement being executed. It is not very clear as to why this rarely needed error-prone behavior was made the default behavior in Java and C++, and C# doesn't do this.

20.21.16 Predefined Types

The C# primitive types are basically the same as the ones in Java except they add unsigned types to. We have got sbyte, byte, short, ushort, int, uint, long, ulong, char, float & double. The only surprise here is the 12-byte "decimal" floating point number which can take advantage of the latest processors.

20.21.17 Field Modifiers

Again, the field modifiers are basically the same as the ones in Java. To express fields which can't be modified, C# uses the const and readonly modifiers. A const field modifier is like the Java final field modifier, and compiles so that the actually value is part of the IL code. A readonly modifier compiles so that at run-time the value is evaluated. This allows an upgrade, say to one one of the standard C# libraries, without breaking your deployed code.

20.21.18 Jump Statements

They are in reality a distant relative of the evil goto statement we remember from basic 20 years ago. A goto statement must point to a label or one of the options in a switch statement. The first usage of pointing to a label is similar to the usage of a continue label statement in Java, except with a little more freedom. Such a goto statement may point anywhere within its scope, which restricts it to the same method, or finally block it if it is declared within one. It may not jump into a loop statement which it is not within, and it cannot leave a try block before the enclosing finally block(s) are executed. The continue statement in C# is equivalent to the continue statement in Java, except that it can't point to a label.

20.21.19 Assemblies, Namespaces & Access Levels

In C#, you can organize the components of your source-code (classes, structs, delegates, enums, etc.) into files, namespaces & assemblies.

Namespaces are nothing but syntactic sugar for long class names. For instance, rather than calling a class Genamics.WinForms.Grid you can declare the class as Grid and enclose the class with:

```
namespace Genamics.WinForms {
    public class Grid {
        ....
    }
}
```

For classes which use Grid, you can import it using the 'using' keyword instead of referring to its full class name Genamics.WinForms.Grid.

Assemblies are .exes or .dlls generated from compiling a project of files. The .NET runtime uses the configurable attributes and versioning rules built into assemblies to greatly simplify deployment—no need to hack the registry—just copy the assembly into a directory and it goes. Assemblies also form a type-boundary to deal with type-name collisions, to the extent that multiple versions of an assembly can co-exist in the same process. Each file can contain multiple classes and multiple namespaces. A namespace may also be spread across several assemblies, so there is high level of freedom with this system.

There are five access levels in C#, private, internal, protected, internal protected and public. Private and public have the same meanings as in Java, noting that in C# the default access level is private, rather than package. Internal access is scoped to assemblies rather than namespaces (which would be more analogous to Java). Internal protected access is equivalent to Java's protected access, and protected access is equivalent to Java's private protected access made obsolete in Java some time ago.

20.21.20 Pointer Arithmetic

Pointer arithmetic can be performed in C# within methods marked with the unsafe modifier. When pointers point to garbage collected objects, the compiler enforces the use of the fixed word to pin the object. This is because garbage collectors rely on moving objects around to reclaim memory, but if this happens when you are dealing with raw pointers you will be pointing at garbage. The choice of the word 'unsafe' is well-chosen since it discourages developers from using pointers unless they really need to.

20.21.21 Rectangular Arrays

C# allows both jagged and rectangular arrays to be created. Jagged arrays are much the same as Java arrays. Rectangular arrays allow a more efficient and accurate representation for certain problems. An example of such an array would be:

Listing 20.10

```
int [,,] array = new int [3, 4, 5];
// creates 1 array int [1,1,1] = 5; Using jagged arrays:
int [][][] array = new int [3][4][5];
// creates 1+3+12=16 arrays int [1][1][1] = 5;
```

In combination with structs, C# can provide a level of efficiency making it a good choice for areas such as graphics and mathematics.

20.21.22 Constructors and Destructors

```
You can specify optional constructor parameters:
class Test {
    public Test () : this (0, null) {}
    public Test (int x, object o) {
    }
}
```

You can specify static constructors:

```
class Test {
    static int[] ascendingArray = new int [100];
    static Test () {
        for (int i = 0; i < ascendingArray.Length; i++)
            ascendingArray [i] = i;
    }
}
```

Destructors are specified with the C++ naming convention using the ~ symbol. Destructors can only be made for references types and not value types, and cannot be overridden. A destructor can not be explicitly called, since it doesn't make much sense to do so when an object's lifetime is managed by the garbage collector. Each destructor in an object's hierarchy (from the most derived to the least derived) is called before the memory for the object is reclaimed.
Despite the naming similarity with C++, destructors in C# are much more like Java finalizers. This is because they are called by the garbage collector rather than explicitly called by the programmer. Furthermore, like Java finalizers, they cannot be guaranteed to be called in all circumstances (this always shocks everyone when they first discover this). If you are used to programming with deterministic finalization (you know when an object's destructor is called), then you have to adapt to a different model when moving to Java or C#. The model Microsoft recommends and implements throughout the .NET framework is the dispose pattern, whereby you define a dispose() method to for classes which manage foreign resources such as graphics handles or database connections. For distributed programming, the .NET framework provides a leased based model to improve upon DCOM reference counting.

20.21.22 Managed Execution Environments

The comparison between [C#/IL Code/CLR] and [Java/Byte-Code/JVM] is an inevitable and valid comparison to make. Let us make these comparisons after explaining why these technologies were made in the first place.

With C and C++ programs, you generally compile the source code to assembly language code which will only run on a particular processor and a particular OS. The compiler needs to know which processor it is targeting, because processors vary in their instruction sets. The compiler also needs to know what OS it is targeting, because OSs vary in ways such as how executables

work and how they implement some basic C/C++ constructs such as allocating memory. This C/C++ model has been very successful (most software you're using probably has been compiled like this), but has its own limitations. These limitations are as follows:

1. It does not provide a program with a very rich interface to interact with other programs (Microsoft's COM was built to address this limitation);
2. It does not allow a program to be distributed in a form which can be used as is on different platforms;
3. It does not allow a program's execution to be limited to a sand box of safe operations ;

Java addressed these problems in the same way Smalltalk had done before it by having Java compile to byte-code which then runs on a virtual machine. Byte-code maintains the basic structure of a program before it is compiled, thus making it possible for a Java program to richly interact with another program. Byte-code is also machine-independent, which means the same class file can be used on a number of platforms. Finally, the fact that the Java language did not have explicit memory manipulation (via pointers) made it well suited to writing sand-boxed programs.

Originally, virtual machines had an interpreter which converted a stream of byte-code instructions to machine code on the fly, but this dreadfully slow process was never appealing to the performance conscious programmer. Today most Java Virtual Machines use a Just-In-Time compiler which basically compiles to machine-code class skeletons just before they enter scope and method bodies just before they are executed. It is also possible to convert a Java program to assembly language before it runs to eliminate the overhead in start-up time and memory of a Just-In-Time compiler.

As with compiling a Visual C++ program, this process does not necessarily remove the program's dependence on a runtime. The Java runtime (also covered by the term "Java Virtual Machine") will handle many vital parts of the programs execution such as garbage collection and security. This runtime is referred to as a managed execution environment.

Though somewhat obscured by terminology, the basic model .NET uses is the same as described above, though it was built to never use intepreting. Important improvements .NET will offer will come from the IL design itself, since the only way Java can match such improvements is by changing the byte-code specification which would generate serious compatibility issues. Details of these improvements can be appreciated only by developers who understand both byte-code and IL-code.

Here are some of the design decisions for IL which are intended to improve upon byte-code:

1. To provide greater type neutrality (helps implementing templates) *To provide greater language neutrality
2. To always be compiled to assembly language before executing, and never interpreted
3. To allow additional declarative information to be added to classes, methods, etc., see Chapter 15 (Attributes)
4. Currently, the CLR has yet to provide multi-OS support, but provides greater interoperability to JVMs in other areas.

20.21.23 Libraries

A language is pretty much useless without libraries. C# has remarkably few core libraries, but utilizes the libraries of the .NET framework (some of which were built with C#). This article is mainly concerned with addressing the C# language in particular, rather than on .NET, which is best dealt with in a separate article. Briefly, the .NET libraries come with a rich set of libraries including Threading, Collection, XML, ADO+, ASP+, GDI+ & WinForms libraries. Some of these libraries are cross-platform, while others are Windows dependent, but see the next section for a discussion on platform support.

20.21.24 Interoperability

One may think that it would be useful to group interoperability into three divisions: Language interoperability, Platform interoperability, and Standards interoperability. While Java has its defining strength in platform interoperability, C# has it's strength in language interoperability. Both have strengths and weaknesses in standards interoperability.

Language Interoperability:

1. This is the level and ease of integration with other languages. Both the Java Virtual Machine and the Common Language Runtime allow you to write code in many different languages, so long as they compile to byte code or IL code respectively. However, the .NET platform has done much more than just allow other languages to be compiled to IL code. NET allows multiple languages to freely share and extend each others libraries to a great extent. For instance, an Visual Basic programmer could import a C# class, override a virtual method of that class, and the C# object would now use the Visual Basic method (polymorphism). In case you were wondering, VB.NET has been massively upgraded (at the expense of compatibility with VB6) to have modern object oriented features.
2. Languages written for .NET will generally plug into the Visual Studio.NET environment and use the same RAD frameworks if needed, thus overcoming the "second rate citizen" effect of using another language.
3. C# provides P/Invoke, which is a much simpler (no-dlls) way to interact with C code than Java's JNI. This feature is very similar to J/Direct, which is a feature of Microsoft Visual J++.

Platform Interoperability:

1. Generally this means OS interoperability, but over the last few years the internet browser has emerged as a platform in itself.
2. C# code runs in a managed execution environment, which is the most important technological step to making C# run on different operating systems. However, some of the .NET libraries are based on Windows, particularly the WinForms library which depends on the nitty gritty details of the Windows API. There is a project to port the Windows API to Unix systems, but this isn't here now and Microsoft has not revealed any firm indication of their intentions in this area.

However, Microsoft hasn't ignored platform interoperability. The .NET libraries provide extensive capabilities to write HTML/DHTML solutions. For solutions which can be implemented with a HTML/DHTML client, C#/.NET is a good choice. For cross-platform projects which require a more complex client interface, Java is a good choice. Kylix, a version of Delphi which allows the same code to compile to both Windows and Linux may also be a good choice for rich cross-platform solutions in the future.

Microsoft has submitted the C# specification as well as parts of the .NET specification to the ECMA standards body.

Standards Interoperability:

1. These are all the standards like databases systems, graphics libraries, internet protocols, and object communication standards like COM and CORBA, that the language can access. Since Microsoft play a big role in defining many of these standards, they are in a very good position to support them.
2. C# objects, since they are implemented as .NET objects, are automatically exposed as COM objects. C# thus has the ability to expose COM objects as well as to use COM objects. This will allow the huge base of COM code to be integrate with C# projects. .NET is a framework which can eventually replace COM—but there is so much deployed COM code that by the time this happens .NET will be replaced by the next wave of technology.

21

Interfacing With Com

21.1 Introduction

Component Object Model and Distributed Component Object Models have been in use for over two years now. Any knowledge of COM is incomplete without at least the basics of plug-in and shared data techniques.This chapter describes these and some other relevant aspects of COM.

The COM model allows developers to build real, stand-alone components. Users (clients) of COM components (servers) can utilize pre-built functionality in server objects without intimate knowledge of the server. COM, instead of trying to link in a component's functionality at design time or provide a path to the component in your source code, COM uses the CLSID to determine the binary's location—whether it is local or remote.

Interfaces are represented by GUIDs. Interface IDs, known as IIDs are in reality GUIDs which are used to determine the available methods. Accordingly, if you want to use the same Interface to represent methods in your component, it must have exactly the same signature as the original interface. This is the concept of immutability.

As long as the mapping mechanism knows where to find the server component and the client knows the methods of the interface, the application is ready to run. When you type in the code:

```
CoCreateInstance(CLSID,_,IID,_);
```

Just consider that you are asking a mapping utility (the Registry) to locate a binary (DLL or EXE) identified by CLSID, and to load it into memory with LoadLibrary or CreateProcess to determine whether the binary supports a prescribed set of methods.

In general, when you want to use a component, you call CoCreateInstance to load the binary identified in the mapping utility. See if it supports the unique signature identified by IID, and if it does so, call the methods as you would on any other object.This concept is the most basic feature of COM.

Because any component can create a signature identified by an IID, the COM model allows clients to interchange components either by changing the CLSID's binary entry locations in the registry (possible but not recommended) or by allowing client applications to select from a variety of binaries supporting the same interface at run time to choose desired functionality (highly recommended).

This concept of 'interchangeability because of interface immutability' will be emphasized in this chapter.

Why is the concept of immutable interfaces so important? If we popularize interfaces among developers, we cannot only utilize each other's components but also interchange them when better widgets are developed .

One of today's most popular immutable interfaces is the ADORecordset championed by Microsoft. Any Windows database developer can work with data given to him in the form of an ADORecordset. However, here we are not going to work with ADORecordsets—we are going to emphasize the concept of immutability and attempt to popularize the use of immutable interfaces through custom component categories in the registry.

21.2 What We Will Build?

Rather than supporting a Connection Point—which is preferably used for scripting clients —we will create a straight call back interface. Additionally, in the DCOM server, we will demonstrate the technique for sharing data between different COM instances without using singletons. Data between COM instances cannot be shared without singletons.This concept will be demonstrated in Part 1.

In this chapter, we will build a couple of plug-in COM objects that support the same interface through C++ class inheritance in Active Template Library (ATL). Using C++ inheritance, we will demonstrate how to use base class methods in derived COM objects. This technique allows you to create interfaces without having to implement every method of the interface; one of the most marked advantages of combining a language that supports inheritance (such as C++) with the COM's concept of immutability. This section will also demonstrate custom registration using the .rgs files that are automatically created through the ATL wizards. With a few simple lines, you can create custom registration entries for your components without bothering about registry files or Win32 API calls.

Lastly, we will integrate all these with a Microsoft Foundation Classes (MFC) dialog application. The dialog application will use the DCOM server, implement the call back component for the DCOM server, and allow the user to dynamically change plug-ins to process the data.

To build the samples, you will need Visual C++ 6.0. While you can utilize Windows 95, Windows 98, or Windows NTr as your development platform, Windows NT 4.0 Workstation is recommended for any serious DCOM work. Using Windows NT, you can examine the processes and shut things down with a little bit more stability and ease than in Windows 95 or Windows 98.

21.3 Creating Plug-ins

In step 2, we are going to create plug-in data processors. The purpose of a data processor is to message or convert data. The purpose of a data processor plug-in is to dynamically change the data messaging or conversion technique. For example, our client application will provide a list of currencies and request dollar equivalents. Our server only knows how to return values in dollars. (Let's assume that we do not have the source code for the DCOM server.) By introducing the concept of a plug-in in our client application, we can convert the returned value into a different currency valuation on the fly. With a plug-in model, we can dynamically add new currency converters without recompiling any code, allowing us to convert in any direction between

Dollars, Yen, and Marks today—and then to add Lira and Pesos to our system, without touching the client. Now that you understand a plug-in model, let's write some plug-ins.

The first requirement is to create an interface for our data processor. This interface will serve as the immutable signature for all data processors in this example. We will also create a C++ class that will implement the interface method so that our derived data processors can optionally implement the interface method. No, we are not violating the COM rule that says one must provide an implementation of every method of the interface, but rather leveraging C++ to provide the implementation in a base class if a derived class chooses not to implement a method. Since COM methods are virtual, method implementation location (base or derived) will be determined at run time. This is a great advantage of developing COM in C++. If you had an interface that contained 100 methods, and you desire to override only one method,it can very well be done with C++ and COM.

The base interface and C++ class are included with the sample code in the BaseClassDataProcessor folder, and are shown below (first, the IDL file):

Listing 21.1

```
// This file will be processed by the MIDL tool to contain the
// definition of the IBaseClassDataProcessorObj Interface.
import "oaidl.idl";
import "ocidl.idl";
[
  object,
  uuid(FFD77865-53CF-11D2-9195-006008A4FB73),
  helpstring("IBaseClassDataProcessorObj Interface"),
  pointer_default(unique)
] interface IBaseClassDataProcessorObj : IUnknown {
  [helpstring("method ProcessData")]
  HRESULT ProcessData([in] long lValue, [out,retval] float*
   pfloatResult);
};
```

And the header file:

```
#ifndef __BASECLASSDATAPROCESSOROBJ_H_
#define __BASECLASSDATAPROCESSOROBJ_H_
class CBaseClassDataProcessorObj : public IBaseClassDataProcessorObj
{
  protected:
  virtual bool VerifyData(long lValue)
   {
    if(lValue < 0)
    {
      _ASSERTE(false);
      return false;
    }
    return true;
   }
  STDMETHOD(ProcessData)(long lValue, float* pfloatResult)
   {
    bool bValid = VerifyData(lValue);
    if(!bValid)
    {
      return E_INVALIDARG;
    }
```

```
    }
    *plResult = lValue;
    return S_OK;
  }
};
```

The CBaseClassDataProcessorObj implements IBaseClassDataProcessorObj::ProcessData. The class also provides a check to ensure the value supplied to it is not less than 0 through the method verify data. (It is impossible for a currency to have an exchange rate less than 0.)

In addition to supplying an interface definition and a default class implementation, we need to supply the definitions of the interfaces, which is done by processing the IDL file through the MIDL compiler.The MIDL compiler will generate a new file called BaseClassDataProcessor.h. This file, which will be included in the client application, provides the function prototype for the IBaseClassDataProcessor::ProcessData.

The easiest way to process a stand-alone IDL through the MIDL compiler is within a command window (DOS). You may issue the command as follows:

```
midl BaseClassDataProcessor.idl
```

(You may need to run vcvars32 bat, which is located in your VC98\bin folder, from the command window in order to set the environment variables so the MIDL compiler can be found.)
Once the MIDL step is complete, you are ready to create some derived data processors. The first data processor we will create will be the YenProcessor. Using AppWizard, create a new ATL COM AppWizard project as before—but this time, select Server Type Dynamic Link Library (DLL).

Next, we need to insert a new ATL Object with the wizard. Add an object name, YenProcessorObj, and select the default attributes. Since the COM object will be created inside an MFC application, single-threaded apartment (the default on the Attributes tab) is perfect.

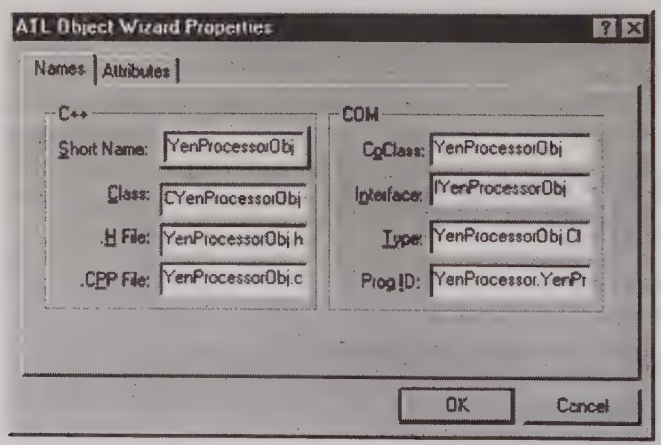


Fig.1: Inserting a new ATL Object

Since we are going to use the IBaseClassDataProcessorObj interface method, there is actually no need to create a new interface for this COM object. Accordingly, open up the wizard-generated IDL file, import the IBaseClassDataProcessorObj interface, remove the wizard-generated interface, and change the default interface in the coclass section to IBaseClassDataProcessorObj. The YenProcessorObj IDL code now declares only the IBaseClassDataProcessorObj interface. The type library is generated by the MIDL compiler now supports only IBaseClassDataProcesssorObj. (You can look at the type library with OLEView, which comes with Visual C++ 6.0).The complete YenProcessor.idl code is listed below:

Listing 21.2

```
// YenProcessor.idl : IDL source for YenProcessor.dll
// This file will be processed by the MIDL tool to
// produce the type library (YenProcessor.tlb) and marshalling code.
import "..\BaseClassDataProcessor\BaseClassDataProcessor.idl";
[
  uuid(9B7CB567-5635-11D2-9196-006008A4FB73),
  version(1.0),
  helpstring("YenProcessor 1.0 Type Library")
]
  library YENPROCESSORLib {
  importlib("stdole32.tlb");
  importlib("stdole2.tlb");
  [
   //this is the components CLSID
   uuid(9B7CB574-5635-11D2-9196-006008A4FB73),
   helpstring("YenProcessorObj Class")
  ]
  coclass YenProcessorObj
  {
   [default] interface IBaseClassDataProcessorObj;
  };
};
```

We now need to modify our YenProcessorObj so that it supports the IBaseClassProcessorObj interface. This requires four steps in YenProcessorObj.h:

Listing 21.3 :

Step 1: Replace the default inheritance

```
public IDispatchImpl_ with
public CBaseClassDataProcessorObj This allows us to inherit the default
implementation of class CBaseClassDataProcessorObj.

Step 2: Replace the default COM Map interface entries of
  COM_INTERFACE_ENTRY(IYenProcessorObj)
  COM_INTERFACE_ENTRY(IDispatch)
with
  COM_INTERFACE_ENTRY(IBaseClassDataProcessorObj) This changes our COM
object from supporting the wizard-supplied IYenProcessorObj and IDispatch
to support only IBaseClassDataProcessorObj.
```

Step 3: Provide an implementation for IBaseClassDataProcessor::ProcessData (shown below along with a complete listing of YenProcessorObj.h).

```
#define __YENPROCESSOROBJ_H_ #include "resource.h"     // main symbols
#include <BaseClassDataProcessorObj.h>
// CYenProcessorObj class ATL_NO_VTABLE CYenProcessorObj :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CYenProcessorObj, &CLSID_YenProcessorObj>,
  public CBaseClassDataProcessorObj
{ public:
  CYenProcessorObj() {}
DECLARE_REGISTRY_RESOURCEID(IDR_YENPROCESSOROBJ)
DECLARE_PROTECT_FINAL_CONSTRUCT() BEGIN_COM_MAP(CYenProcessorObj)
  COM_INTERFACE_ENTRY(IBaseClassDataProcessorObj)
END_COM_MAP()
// IYenProcessorObj public:
  STDMETHOD (ProcessData) (long lValue, float* pfloatResult)
  {
   bool bValid = VerifyData(lValue);
   if ( !bValid )
   {
     return E_INVALIDARG;
   }
   *pfloatResult = ((float)lValue) * (float).1111;
   return S_OK;
  }
};
#endif //__YENPROCESSOROBJ_H_
```

Step 4: Add #include <BaseClassDataProcessorObj.h> to the class declaration, and add ..\BaseClassDataProcessor to the Additional include directories, as shown below:

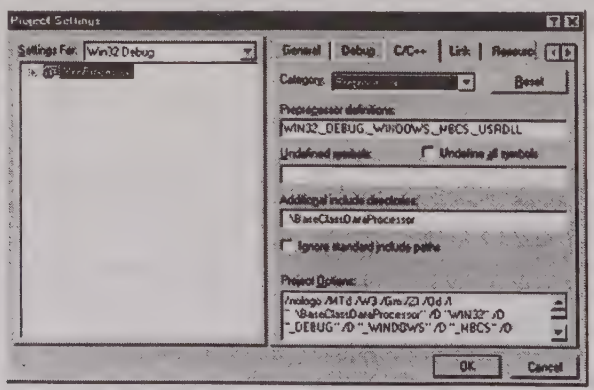


Fig. 2: The project settings

The final item for the YenProcessor is to consider how it should be made available to our client application. As you know, in order to use a COM object in a client, the client must be aware of the CLSID, IID, and interface method prototypes. Since all our data processors will support interface IBaseClassDataProcessorObj, the client can have IID—IBaseClassDataProcessorObj hard-coded as a constant and include the method prototypes in BaseClassDataProcessor.h (generated by the MIDL compiler when it processed BaseClassDataProcessor.idl). However, since we want the client application to dynamically discover new data processors, hard-coding CLSIDs would be inappropriate.

The solution is to create a 'Component Category' in the registry for our data processors. Our client application (or any client application) can, at run time, find all available data processors and allow the user to select which component to utilize. Accordingly, we need to create a component category that lists the CLSIDs of data processors. It will also be helpful if we associate an easy-to-understand name with the CLSID, so that a user can easily identify the component.

An easy way of doing this is by using a script that is linked into the binary, rather than writing a separate registry file. Just add the following lines to the YenProcessorObj.rgs code. This way, when the COM object is registered on a machine, the component category will be created.

```
HKLM { SOFTWARE {
  PantherCurrencyComponents = s 'PantherCurrencyComponents'
  {
   AvailablePlugInDataProcessors
   {
val Yen = s '{9B7CB574-5635-11D2-9196-006008A4FB73}'
   }
  }
}
}
```

The code above adds a key under HKEY_LOCAL_MACHINE\Software called PantherCurrencyComponents. Under the PantherCurrencyComponents key, it adds another key titled AvailablePlugInDataProcessors. Under this key, it adds a name entry called Yen with a string value set to the CLSID of the component (the CLSID is in the coclass section of the IDL file).

The screen shot below shows the registry with three available currency data processors, including the Dollar processor that =demonstrates using the CBaseClassDataProcessorObj implementation of IBaseClassDataProcessorObj:: ProcessData, since the value returned from the DCOM server returns dollar equivalents.

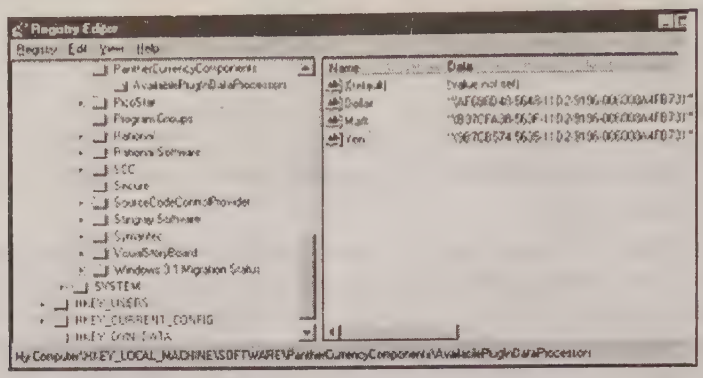


Fig. 3: The registry, with three currency data processors

21.4 The Client Application

Now that we have completed the DCOM server and have created three dynamically interchangeable plug-ins, it is time to tie them all together in a client application. As noted in the beginning of Part 2, the client application will provide a list of currencies and request dollar equivalents from the DCOM server. When the DCOM server processes the request, it returns the dollar conversion value to the client through the client's ICallBackObj interface.

To accomplish this, the client application is the responsible for creating the ICallBackObj interface and handing it to the DCOM server. In Part 1, we included CallBackObj.idl in the DCOM server's IDL file, so that the DCOM server would have intimate knowledge of the interface and the ICallBackObj interface could be passed as a signature type to the DCOM server. We could have generated the ICallBackObj methods with the MIDL compiler separately from the DCOM server application and passed an IUnknown* to the DCOM server. The DCOM server would then have to QueryInterface for IID_ICallBackObj. This methodology would work effectively, and is very similar to using Connection Points. However, passing an IUnknown* interface and querying for IID_ICallBackObj results in a run-time error if the passed interface does not support ICallBackObj. The methodology used in our sample results in a compile-time error if any interface other than an ICallBackObj is passed in IDCOMServerObj::RegisterCallBack.A general rule is to use Connection Points in situations where the client may be implemented in script; use a proprietary interface when you desire a proprietary call back.

Before we proceed and implement the ICallBackObj in the client and process the results returned from the DCOM server, let's review the required functionality in the client. Once the client receives the dollar conversion value from the server, we also want to show the originally selected currency type converted into a third currency (currency traders call this cross trading). Accordingly, we need to be able to select the source currency as well as the target currency and load the plug-in data converter for the target currency. After these, our client application should look like the following application :

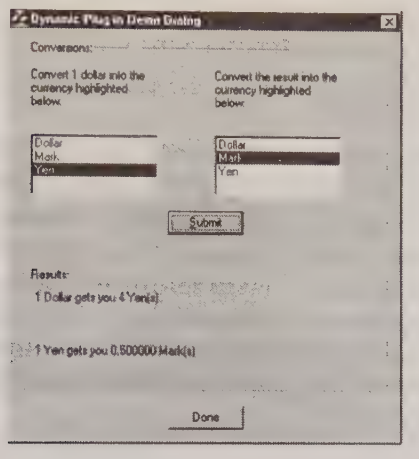


Fig. 4: The currency conversion application

We need to generate a dialog-based application using the MFC Application Wizard. In the samples that come with the article, we created a dialog-based application and assigned 'ClientApp' as the application name. The other wizard defaults are fine.

Once the application is generated, we need to obtain a connection to the DCOM Server. Setting up a client application for connecting to an ATL-based COM server requires the following steps:

Make available the CLSID of the COM server and the interface IDs that we desire to utilize Provide a way for the client application to find the method declarations of the interface methods Include the ATL related files and initialize ATL related variables Steps 1 and 2 involve including a series of files in the projects and making sure the project can locate the files that are included. This is accomplished by setting 'Additional include directories' under menu item Project Settings | C++ | Category Preprocessor, as shown in the screen shot below. In addition to including the relative path to the DCOM server, we also include the relative path to the BaseClassDataProcessor folder, since we will eventually need the method declarations for IBaseClassDataProcessorObj.

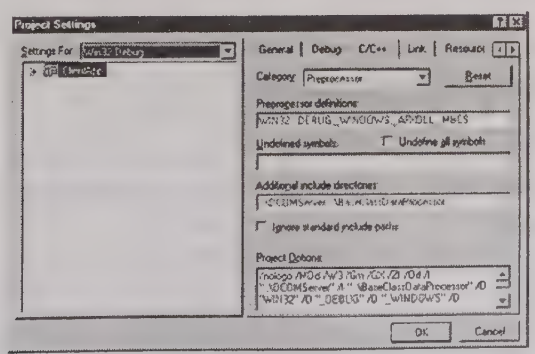


Fig.5: Adding the relative path

Making the CLSID and IIDs available is now quite easy. At the top of ClientAppDlg.cpp, we simply add the lines:

//midl generated code with interface prototypes #include <DCOMServer.h> //CLSID and IID #include <DCOMServer_i.c> That provides us with the interface method prototypes and the CLSIDs and IIDs of the DCOM server.

In order to add the ATL-related declarations and variables in stdafx.h, we add the following lines that declare all our ATL-related variables and function declarations.

//Add the includes for ATL and atlimpl.cpp since there is //there is no ATL lib file. #include <atlbase.h> extern CComModule _Module; #include <atlcom.h> #include <atlimpl.cpp> The last step is to initialize the CComModule variable that is declared in stdafx.h above. Inside the application module, CClientApp.cpp, we create stdafx.h's externally declared _Module variable and initialize it. Also, even though the wizard-provided defaults enabled the application as an ActiveX Control Container, they did not provide us with the one line required to initialize COM. This very important line is AfxOleInit().

```
//———————————————————— -CComModule _Module;
// CClientAppApp initialization                          //////// //
BOOL CClientAppApp::InitInstance() {
  AfxOleInit(); //initialize OLE
  AfxEnableControlContainer();
  //ATL loading _Module.Init(NULL, NULL);
—
—
```

We are now ready to implement the ICallBackObj in the client. Our C++ class representation of ICallBackObj will be CCallBackObj. Because ICallBackObj derives from IDispatch, the CCallBackObj class needs to inherit not only from CComObjectRoot (which handles reference counting) but also from IdispatchImpl, which provides a default implementation for the IDispatch portion of the interface on this COM object.

In addition to providing the implementation of ICallBackObj::ReceiveDataFromServer, we need to provide one more method, SetOwnerWindow. When we receive data from the DCOM Server, the data is being sent on a thread that belongs to the DCOM server. Since we are using MFC, data needs to be processed on the main thread that hosts the MFC GUI objects. In order to move data from the DCOM server's worker thread back to our application's main thread, we have the ICallBackObj::ReceiveDataFromServer method post a message to the application's main thread with the data as the message's LPARAM. The complete source for CallBackObj.h is listed below; pay special attention to the ReceiveDataFromServer method:

Listing 21.4

```
#ifndef __CALLBACKOBJ_H_ #define __CALLBACKOBJ_H_
#include "resource.h"     // main symbols
#include <DCOMServer.h>   //midl generated code
class CCallBackObj :
  public CComObjectRoot,
  public IDispatchImpl<ICallBackObj, &IID_ICallBackObj,
          &LIBID_DCOMSERVERLib>
{
```

```
public:
  CCallBackObj()
  {
   m_pOwnerWnd = NULL;
   m_lLastDataReceivedFromServer = -1;
  }
  void SetOwnerWindow(CWnd* pOwnerWnd)
  {
   m_pOwnerWnd = pOwnerWnd;
  }
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP(CCallBackObj)
  COM_INTERFACE_ENTRY(ICallBackObj)
  COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
protected:
  CWnd* m_pOwnerWnd;
  long m_lLastDataReceivedFromServer;
public:
  STDMETHOD(ReceiveDataFromServer)(/*[in]*/ long lData)
  {
   AFX_MANAGE_STATE(AfxGetStaticModuleState())
   if(m_pOwnerWnd != NULL &&
     m_pOwnerWnd->GetSafeHwnd())
   {
     //post a message since we are on a different thread
     //which will put us back on the main thread
     m_lLastDataReceivedFromServer = lData;
     m_pOwnerWnd->PostMessage(
WM_RECEIVE_DATA_FROM_PLUG_IN,
          0, m_lLastDataReceivedFromServer);
  }
  return S_OK;
} };
```

#endif//__CALLBACKOBJ_H_

Our next step is to connect to the server, create the ICallBackObj interface, set the owner window, and pass the ICallBackObj to the server. The correct place to do this is in OnInitDialog, since we are using class member variables to hold the COM objects. The source code from OnInitDialog that does this is as shown here:

Listing 21.5

```
IUnknown*   pUnk    =    NULL;    HRESULT    hRes    =
::CoCreateInstance(CLSID_DCOMServerObj, NULL, CLSCTX_ALL,
        IID_IUnknown, (void**)&pUnk);
if(SUCCEEDED(hRes))
   //get our IDCOMServerObj into our class member
  hRes = pUnk->QueryInterface(IID_IDCOMServerObj,
(void**)&m_pIDCOMServerObj);
```

```
(void**)&m_pIDCOMServerObj);
  pUnk->Release();
  pUnk = NULL;
  ASSERT(SUCCEEDED(hRes));
  ASSERT(m_pIDCOMServerObj != NULL);
  if(SUCCEEDED(hRes))
  {
   //create the CCallBackObj and set the owner window
   hRes =
     CComObject<CCallBackObj>::CreateInstance(
      &m_pIServerCallBack);
  ASSERT(SUCCEEDED(hRes));
   if(SUCCEEDED(hRes))
   {
     m_pIServerCallBack->SetOwnerWindow(this);
     //registher the interface with the DCOM Server
     hRes = m_pIDCOMServerObj-RegisterCallBack(
m_pIServerCallBack);
     ASSERT(SUCCEEDED(hRes));
   }
  }
}
```

The next to-do item is to obtain the CLSIDs of available plug-ins that support IBaseClassDataProcessorObj for currency conversions. This is also done in OnInitDialog, but rather than instantiating the COM objects at this time, we will place the currency names in a list box for display and their CLSIDs as data items in the list box entries. Here is the code that does this:

```
/*
```

Fill the list box with available plug-ins.

Each plug-in has an easy to use name and a CLSID as string. All plug-ins support the same interface.

The easy to use name goes in the list box to display the CLSID as string goes in the list box's item data.

Listing 21.6

```
*/
BOOL bCLSID = FALSE;
CString sKey =
_T("Software\\PantherCurrencyComponents\\AvailablePlugInDataProcessors\\");

//first open the registry key
HKEY hKey = NULL;
LONG lRes = ::RegOpenKeyEx(HKEY_LOCAL_MACHINE, sKey, 0,
       KEY_READ, &hKey);
```

```
    if (lRes == ERROR_SUCCESS && hKey != NULL)
    {
     DWORD dwIndex = 0; ' DWORD size;
     TCHAR buff[128];  LONG lRes;
     DWORD type;  byte buff2[128];
     DWORD size2 = 128;
     //Read registry entries until the end of key is reached
     while(TRUE)
     {
       buff[0] = '\0';
       size = 128;
       lRes = ::RegEnumValue(hKey, dwIndex++, buff, &size,
            NULL, &type, buff2, &size2);
       if(lRes == ERROR_SUCCESS)
       {
        CString sKeyName = buff;
        int iIndex = m_ListBoxWithCLSIDs.AddString(sKeyName);
        CString* psClsid = new CString(buff2);
        m_ListBoxWithCLSIDs.SetItemData(iIndex,
(DWORD)psClsid);
        CString sTemp = *psClsid;
        sTemp += _T("\n");
        TRACE(sTemp);
       }
       else
       {
        break;
       }
     }

     // now close the registry key
     ::RegCloseKey(hKey);
     hKey = NULL;
    }
    if(m_ListBoxWithCLSIDs.GetCount())
     m_ListBoxWithCLSIDs.SetCurSel(0);
```

Now, let us see what occurs when data is received from the DCOM Server. As we saw earlier, ICallBackObj::ReceiveDataFromServer posts a WM_RECEIVE_DATA_FROM_PLUG_IN message back to this dialog. The dialog picks up the message vis-...-vis the message-mapped function OnMessageFromPlugIn.

The LPARAM of the message is the actual dollar conversion rate (long value) received from the DCOM server, so it is displayed as is in a static control. But we do want to 'cross trade' the original currency into another currency that may not be Dollars. Accordingly, we need to load the plug-in converter that performs the currency conversion. Since the CLSIDs of the all the conversion plug-ins are already loaded in a list box, and they all support the same interface, all

we need to do is determine which list box entry is highlighted, obtain its data item, covert it to a CLSID, cocreate the COM object, and make a method call. The code that does this is as shown:

- Get index from list box.
- Get string from item data.
- Convert to CLSID
- CoCreateInstance
- Call interface method.

Listing 21.7

```
    iIndex = m_ListBoxWithCLSIDs.GetCurSel();

    if(iIndex < 0)
    {
     ::MessageBox(NULL, _T("Please highlight a plug-in"),
  _T("Message"), MB_OK);
     return -1;
    }
    CString sPlugInCurrency;
    m_ListBoxWithCLSIDs.GetText(iIndex, sPlugInCurrency);

    if(sPlugInCurrency.Find(_T("Dollar")) != -1)
     sPlugInCurrency = _T("Dollar");
    else if(sPlugInCurrency.Find(_T("Mark")) != -1)
     sPlugInCurrency = _T("Mark");
    else if(sPlugInCurrency.Find(_T("Yen")) != -1)
     sPlugInCurrency = _T("Yen");
    else
    {
     ASSERT(false);
    }
   CString* pstrCLSID = (CString*)m_ListBoxWithCLSIDs.GetItemData(iIndex);
    CComBSTR bstr(*pstrCLSID);
    CLSID clsid;

    HRESULT hRes = ::CLSIDFromString(bstr, &clsid);
    if(!SUCCEEDED(hRes))
    {
     ::MessageBox(NULL, _T("Higlighted item does not have a
  valid CLSID"),
                  _T("Message"), MB_OK);
     return -1;
    }
    IUnknown* pUnk = NULL;
    hRes = CoCreateInstance(clsid, NULL, CLSCTX_ALL,
        IID_IUnknown, (void**)&pUnk);
```

```
  if(SUCCEEDED(hRes))
  {
   IBaseClassDataProcessorObj* pIDataProcessor = NULL;
   hRes = pUnk->QueryInterface(IID_IBaseClassDataProcessorObj,
(void**)
                  &pIDataProcessor);
   pUnk->Release();
   if(SUCCEEDED(hRes))
   {
     float fltValueFromDataProcessor;
     hRes = pIDataProcessor->ProcessData(1,
&fltValueFromDataProcessor);
     ASSERT(SUCCEEDED(hRes));
     pIDataProcessor->Release();
     //sTargetCurrency contains target currency
     CString sMessage = _T("1 ");
     sMessage += sTargetCurrency;
     sMessage += _T(" gets you ");
     sValue.Format(_T("%f "), ( ((float)1)/
fltValueFromDataProcessor /
                  ((float)lValueFromServer)));
     sMessage += sValue;
     sMessage += sPlugInCurrency;
     sMessage += _T("(s)");
     pWnd = GetDlgItem(IDC_STATIC_RESULT2);
     pWnd->SetWindowText(sMessage);
   }
   else
   {
     ::MessageBox(NULL, _T("Selected component does not
support
                  IID_IBaseClassDataProcessorObj
 interface"),
        _T("Message"), MB_OK);
   }
  }
```

Other than submitting requests to the server (which is done in CClientAppDlg::OnSubmit()),the client does much more. One of the benefits of interface immutability is that it allows for extensibility through plug-ins that can be created independently of the applications that use them. Your next step is to port these COM objects to Windows CE, move to Chicago, and get a seat of the Chicago Board of Trade.

Panther Software is a software design and consulting company that provides advanced technology solutions to leading software companies and IT divisions of Fortune 1000 companies. Panther can be reached on the Internet at http://www.panthersoft.com/.

21.1.4 Background

FMStocks 2000 uses COM+, not Microsoft Transaction Services (MTS). COM+ is built into Windows 2000, while MTS was an add-on to Windows NTr 4.0. This seems like a minor distinction, but it is very important to developers. Because it uses COM+, developers must make one small change to their applications. With MTS, anytime one object that is in a transaction instantiates another object that must run in the same transaction, the first object's code must use GetObjectContext.CreateInstance to instantiate the new object. This call is required because MTS is not a core part of the operating system. When you instantiate an object using CreateObject, MTS does not know about the object, so the object will run in a new transaction context (if its transaction settings require a transaction).Therefore, you must call CreateInstance to let MTS know you want to create the new object so it can create it inside the same transaction. COM+ changes this because it is part of the operating system and knows what is happening with components and their objects when they are created. In Visual Basic, all you need to do when you want to instantiate an object from another object and have the new object become part of that transaction is use either CreateObject or the New keyword. For instance, you could instantiate the Customer object using either of these two methods:

```
Set oCustomer = CreateObject("Companies.Customer") Set oCustomer = New
Companies.Customer
```

The same approach applies to any object you instantiate, including ActiveXr Data Objects (ADO). However, don't use the New keyword when you dimension the object variable. For instance, if you use the following statement, you will add extra overhead to your application:

```
Dim oCustomer as New Companies.Customer
```

This causes Visual Basic to add code like the following internally every time you use oCustomer:

```
If not isObject(oCustomer) then
    Set oCustomer = New Companies.Customer
oCustomer.Add "John"
```

Every time you use the oCustomer object, Visual Basic must place code before it to make sure it is instantiated. This makes your application code slightly smaller, but adds lots of overhead to your application if you use the object frequently (for instance, in a loop).

21.5 Migrating to COM+

Now, what happens when you migrate an existing application to COM+ and it uses CreateInstance? You should change those CreateInstance statements to CreateObject or use the New keyword. Doing so will improve the performance of your application because it will work directly with COM+ and avoid the overhead of working with CreateInstance while it's really calling CreateObject behind the scenes. Once you have changed CreateInstance to CreateObject, you can take advantage of other COM+ features. You should also create a reference to the COM+ Services Type Library to replace your reference to the MTS type library (see Figure 1). The COM+ Services Type Library supports the same features as the MTS type library, but has new methods that add more functionality. You will still find good old GetObjectContext, SetAbort, and SetComplete there, but you will find many more features as well.

If you look through FMStocks 2000 you will see that SetComplete and SetAbort are used just like they were with MTS. The similarity of the interfaces makes migrating applications easy.

21.6 Managing Components

To put components into COM+ and manage them, you must use Component Services Explorer, which can be found on the Administrative Tools menu. The first thing you must do with a component that is going to run in COM+ is load it into a COM+ application. A COM+ application looks and works just like an MTS package. In fact, you will find that many of the COM+ settings are just like those in MTS. You can create a new application just as you created an MTS package. Right-click the COM+ Applications folder and select New | Application from the submenus. Click Next and you will see the Install or Create a New Application page. Click Create to create a new empty package. On the next page, enter the name for the package.

You can change the server or library setting on this page. While creating a real database service layer (as in DBService), this may be set to Library. The application would then run in the calling process space when called by code in an object that was loaded in another package. This way you can use one package over and over again from code that runs in other packages. This makes it easy to share code between applications, which is what components and objects are for. This setting is on the Create page in COM+, so you don't have to go looking for it after you create the package. You can now easily work with your new COM+ application and load components into it.

Once you have created the new app, just use Windows Explorer to drag the component from its location on the disk and drop it into the application's Components folder, just like you did with MTS. Now the component will run in COM+. Next, open the properties for the objects in the component and tweak their settings. Just like an MTS class, the General property page has a description that you can enter or change at any time.

The Transactions property page for COM+ is similar to the same page in MTS. In MTS, there were four transaction types to choose from. COM+ has similar settings, plus a new one called Disabled. You will notice that the type names are slightly different, but otherwise the first four types match those from MTS. Disabled, the new transaction type, turns off transaction support for a class that just implements logic functions that are not tied to a database and are not transactional. For instance, neither helper classes nor classes that work with the registry need transactional support. You can set the transactional property to Disabled for those classes, and now they won't incur the COM+ transaction overhead. In fact, the documentation states that a class set to Disabled will see no more overhead from COM+ than a standard COM class would.Thus, you get the benefits of COM+ for just-in-time activation without having to incur transactional overhead when you don't need it.

Now, what about Visual Basic and its MTS settings? If you set one of the four MTS settings in Visual Basic, COM+ will pick up the setting from the class when you add that class to an application. You can't select the Disabled setting in Visual Basic. If you set the MTSTransactionMode property to 0_NotAnMTSObject, this will cause the transaction mode property to be set to Not Supported. It will not set it to Disabled. So for now, you must set Disabled outside of the class itself.

The last option on the Transactions page allows you to control the transaction timeout value for a particular class. This is nice because it gives you very granular control. When you have a class that takes a long time to execute its transactions, you can tweak the transaction timeout for

only that class without affecting the settings on the rest of the server.This is handy in Web development since ASP scripts can take a widely variable amount of time to run, especially with fluctuating user demand. You can tweak this timeout for a particular COM+ class without changing the other classes. This should also be useful where you have components that access different databases, where some of the databases are either on a slow server, are slow due to size or database type, or are accessed over a slow link. You can set the timeout higher for the class that uses those databases and leave it at the standard setting for all others.

21.7 Interface Implementation

There's a new feature in COM+ called the IObjectConstruct interface. To use this feature, you use the Implements feature in Visual Basic, allowing you to implement the interface of another class. This is useful when you want to implement a standard interface across several objects. To use IObjectConstruct, you first need to set a reference to the COM+ Services Library in your project. Then to use this interface in your code, use the Implements statement with IObjectConstruct as shown here:

21.8 Implements IObjectConstruct

This should be the first line in your class. Once you have used Implements to bring in the interface, you must implement the interface of the Construct procedure from the IObjectConstruct class. You are responsible for implementing the interface within your own code. In FMStocks 2000, IObjectConstruct is implemented in the DBHelper class. When you implement an interface, your code must provide the implementation (logic) for the entire interface of the Implements class. In this case, you must create a procedure to implement the Construct procedure like so:

```
Private Sub IObjectConstruct_Construct(ByVal pCtorObj As Object)
  'do some stuff here
End Sub
```

When the class containing this code loads, the code fires and you can grab the data from the pCtroObj object that is passed in as a parameter. Let us see where the data in pCtroObj comes from. To use the IObjectConstruct Construct method to extract data from its parameter object, you must set the data somewhere. To do this, you must use Component Services Manager and the Activation property page. Open the properties for the class containing the Construct code and set the string that is passed to pCtroObj administratively by following these steps: In the Details pane of the Component Services administrative tool, right-click the class that you want to configure, then click Properties.

21.9 Click the Activation tab

Click the 'Enable object construction' checkbox. In the 'Constructor string' edit box (see Fig.4), enter the construction string (for example, DSN=AccountingServer). Note that an empty string may be used.

You can put anything you want in the string, and the data does not have to be a DSN. For example, take a look at the Construct code from FMStocks 2000 that's shown in Fig.7. The incoming object (pCtorObj) has a ConstructString that contains the data you entered in the

Activation property page for the class. The code from FMStocks 2000 parses this string and looks for either ConnectionString or Server in the string. If the code finds either one, then it sets the corresponding module level variables. Although it is a pretty simple code, it makes for an effective way of handling connection strings and other variable data that does not lend itself to a database or file.

21.10 More Settings

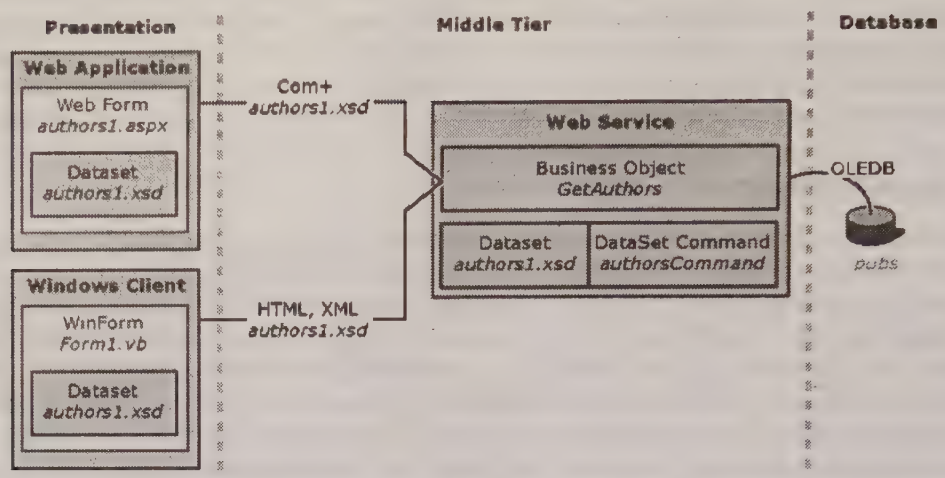
The other items on the Activation tab require some discussion. The Enable Just In Time Activation box is automatically enabled when transactions are required. Checking this box causes COM+ to automatically create and destroy objects as needed to more efficiently manage server resources. This is similar to the way MTS handled activation, but it was automatic in MTS. In COM+, you can easily turn it on and off. This setting is automatically enabled when you use a class that must participate in transactions. The next checkbox, 'Component supports events and statistics,' does just what it says. If you set this box, COM+ will be required to collect information about that class when it is running. Any time you collect information, there is additional overhead. For instance, if you turn on all the auditing features in Windows 2000 or Windows NT, you will make the server slow. Statistics reporting may not create a great deal of overhead for most applications, but you should be aware of the potential to increase resource usage, especially if you want to exploit an application the most. The last checkbox—must be activated in caller's context—forces an object to be executed in the same context as the object calling it.

Conclusion COM+ is a powerful object and transaction management system. The performance numbers from FMStocks 2000 and other tests show that Windows 2000 and COM+ offer unmatched scalability. You can use COM+ to solve many of the maintenance problems you typically face, and Windows 2000 will keep you out of DLL Hell. More and more Web developers are building applications that use COM, and tying those applications together with ASP. With the proper coding techniques and the combination of Windows 2000 and COM+, you can build high-performance applications to serve your Web customers that are stable and easy to maintain.

21.11 Creating a Distributed Application

In this unit, you will create a multitiered distributed application. The application will consist of three logical tiers: Data, Business Objects, and User Interface. The data is a database in SQL Server. The business objects will handle accessing the data and distributing it to the clients. The user interface will consist of both a Web-based application and a traditional rich Windows application.

Architecture of the distributed application you will create is as follows:



The application you will build is a simple data look-up application. You will build both a Windows and Web-based client to display the Authors table from the SQL Server sample Pubs database. For the Web portion, you will use the Web Forms Designer to create HTML that is compatible with a standard HTML 3.2 browser. On the server, the Web Forms code will call a Web Service to retrieve a dataset that contains the requested Authors information from the database. For the Windows portion, you will build a Windows Application, which will talk to this same Web Service to retrieve a dataset containing the author information. The Windows Application will look like any Windows application (launched from the desktop, with menus, and so on) but will communicate to the Web Service using HTTP and XML.

21.12 Process for Creating a Distributed Application

Typically, you create distributed applications one tier at a time, beginning with the data, then the middle-tier business objects, and finally the user interface. The data has already been generated and is available in the Pubs database in SQL Server. Therefore, the walkthrough will start with creating the business object—the Web Service—followed by building the two user interfaces—a Web Forms page and a Windows Form.

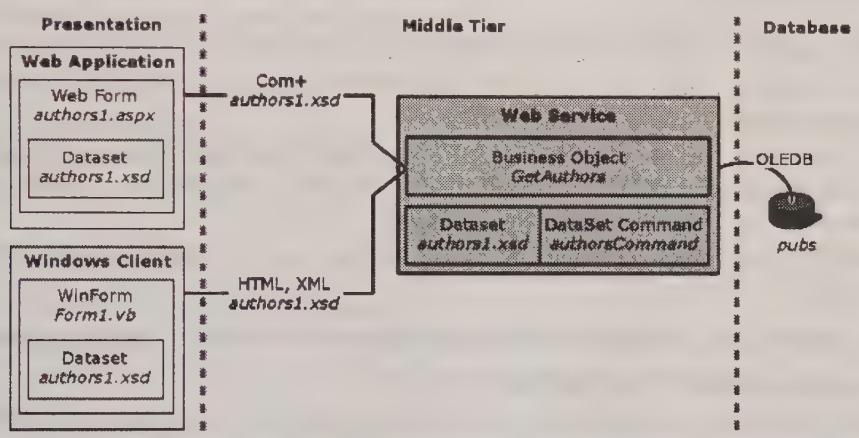
The process for this is:

1. Create the middle-tier business object
   - Create a Web Service project
   - Create a dataset and dataset command
   - Expose the dataset from your business object
2. Create the user interface
   - Create a Windows user interface
   - Create a Web user interface
3. Deploy the solution

21.13 Create the Middle-Tier Business Object

The business object you create will run on a Web server to provide the performance and scalability necessary for a distributed application. In addition, you will expose the business object as a Web Service so that clients can use standard Internet protocols to communicate with your business object from any platform. For details, see Programming the Web with Web Services.

The Web Services project template is a Web-based project designed for creating middle-tier components that expose their interfaces as Web Services. A component will be added to the Web Service that will hold the data connections and dataset. Web Service methods will then be added to the Web Service to expose it, which will make it possible for other applications to view and modify the dataset.



To create a Web Service project

1. On the File menu, point to New, and then click Project to display the New Project dialog box.
2. Depending on what language you want to use, select C# Projects or Visual Basic.NET Projects in the Project Types pane, and then select Web Service.
3. Name the project AuthorsWebService. In the Location box, enter the name of the Web server, http://(server name), and then click OK. Tip If the Web server is on your computer, you can use the server name LOCALHOST.
4. From the Project menu, select Add New Component.
5. Click 'Component' and change the name of the component from the 'Component1' to 'AuthorsComponent' in the Name text box.

In this component you will create a connection to the data store and obtain an instance of the data using a dataset.

21.14 Create and configure a DataSet

Two objects will be added to the AuthorsComponent: a ADODataSetCommand Class and ADOConnection Class. The connection creates a new connection to the database while the dataset command queries the database and feeds the data into the DataSet. The dataset command also updates the database with changes made to the DataSet by the user. Using the connection to the database, a DataSet object will be created to hold an instance of the information present in the database. The DataSet will be used later in the walkthrough to display data in the Windows Form and the Web Forms page. For details, see Introduction to Distributed Applications and Data Integration.

To create a DataSet object, you may proceed as follows:

1. Click Server Explorer. —

Note: By default, Server Explorer appears as a vertical tab on the left of the screen.

2. In Server Explorer, right-click the Data Connections node and select Add Connection from the shortcut menu.
3. In the Connection tab of the Data Link Properties dialog box, enter the name of the SQL Server where the pubs database is installed. If you have SQL Server on the local machine, enter (local).
4. Enter SQL Server logon information. The user you specify must have read/write access to SQL Server.

Tip: Often the user name for the Pubs database is sa and there is a blank password.

5. Select the pubs database from the drop-down list.
6. Click 'Test Connection' to validate the information you provided, and then click OK to establish the connection. [server name].pubs.dbo appears in the Database Connections node of Server Explorer.

Note: If the database connection failed, see your database administrator.

7. In Server Explorer, expand the [server name].pubs.dbo node and then expand the Tables node.
8. Drag the authors table from Server Explorer onto the design surface. The authors table appears in the designer as an ADOConnection paired with an ADODataSetCommand. A connection has now been created to the database, with the transfer of information to be handled by the ADODataSetCommand.
9. From the DataClass menu select Generate DataSet.
10. In the dialog box that appears, name the DataSet Authors1.

The dataset Authors1 has now been created and holds an instance of the Authors table in the Pubs database. The schema for the Authors1 dataset can be accessed in the Authors1.xsd file, which was generated with the dataset.

11. From the DataClass menu select Generate DataSetCommand Methods.

The Code Editor appears, showing that the following methods were generated:

```
Public Sub FillDataSet(ByVal dataset As System.Data.DataSet) Public
Function UpdateDataSource(ByVal updateDataSet As System.Data.DataSet)
As System.Data.DataSet
```

12s. From the File menu, choose 'Save AuthorsComponent'.

21.15 Expose the DataSet from Your Business Object

The next step is to expose the DataSet you just created from your business object. This will make the DataSet available for Windows or Web applications to use.

To add the DataSet to a middle-tier business object

1. In Solution Explorer, double-click AuthorsWebService if it is not already open in the design window.
2. From the View menu, select Code.
3. Add the following Web Service methods:

Listing 21.8

```
'Visual Basic Public Function <webmethod()> GetAuthors() as
webservice1.Authors1
  Dim ds as New Authors1
  Dim dc as New AuthorsComponent()
  dc.FillDataSet(ds)
  Return ds
End Function
Public Function <webmethod()> UpdateAuthors(ByVal ds as
webservice1.Authors1) as webservice1.Authors1
  Dim dc as New AuthorsComponent()
  dc.UpdateDataSource(ds)
  Return ds
End Function // C# [WebMethod] public webservice1.Authors1 GetAuthors()
{
  Authors ds = new Authors1();
  AuthorsComponent dc = new AuthorsComponent();
  dc.FillDataSet(ds);
  return ds;
}
[WebMethod]          public          webservice1.Authors1
UpdateAuthors(webservice1.Authors1 ds) {
  AuthorsComponent dc = new AuthorsComponent();
  dc.UpdateDataSource(ds);
  return ds;
}
```

1. From the Build menu, choose 'Build'.
2. From the File menu, choose 'Save All'.
3. When you are prompted to save the solution file, name it 'AuthorsDistributedApp'.

In these sections, you have created a middle-tier business object that contains a dataset bound to a SQL Server database. You added code to the middle-tier AuthorsWebService Web Service that clients will access to load and update the data in the DataSet.

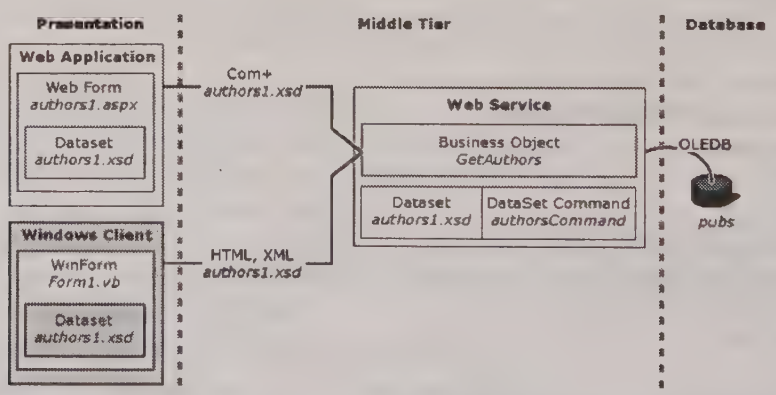
21.16 Create the User Interface

After creating a middle-tier business object for data access and exposing it as a Web Service, the next step is to create the client interfaces. There are two options in this walkthrough: a traditional rich Windows Form and a Web-based form. Both are created in this example as separate projects in the same solution.

21.17 Windows User Interface Option

A Windows interface uses the capabilities of the client machine to handle part of the application processing. Generally, a Windows interface provides greater functionality and a richer user experience than a Web-based interface. There is less of a load on the server than with a Web front end, because the server does not have to perform all of the application logic. Additionally, a Windows interface can take advantage of the resources available through the operating system, including calls to the file system and the Registry.

Windows client in the distributed application is shown as follows:



To create a rich Windows user interface:

1. On the File menu, point to Add, and then click 'New Project' to open the 'Add New Project' dialog box.
2. Select Windows Application and enter 'AuthorsWinClient' for the project name.

The AuthorsWinClient project is added to the solution. Form1 is automatically added to the project and appears in the Windows Forms Designer.

3. In Solution Explorer, right-click the form to rename it 'AuthorsForm'.
4. Add a reference to the Web Service project you created earlier:
   - In Solution Explorer, right-click the 'AuthorsWinClient' project, and select 'Add Web Reference' from the shortcut menu.
   - Enter the location of the .disco file of your Web Service project (http://servername/AuthorsWebService/AuthorsWebService.disco) in the Location box at the top of the 'Add Web Reference' dialog box, and then press ENTER.
   - In the 'Components Found' window, select the components you want and click 'Add Reference'.

You can now bind to the Authors1 dataset generated in the Web Service.

5. If the form is not open in the designer double-click it in 'Solution Explorer'.
6. Click the Data tab of the Toolbox, and drag a 'DataFormWizard' onto the form.

   The 'Data Form Wizard' appears.
7. Choose the Authors1 dataset from the dataset drop-down list, and then click 'Next'. The 'Available Tables' window displays all of the tables contained in Authors1.
8. Select the check boxes to include update and load buttons on your form.

   a. Choose the 'GetAuthors Web Service' method from the Load drop-down list to associate it with the load button.

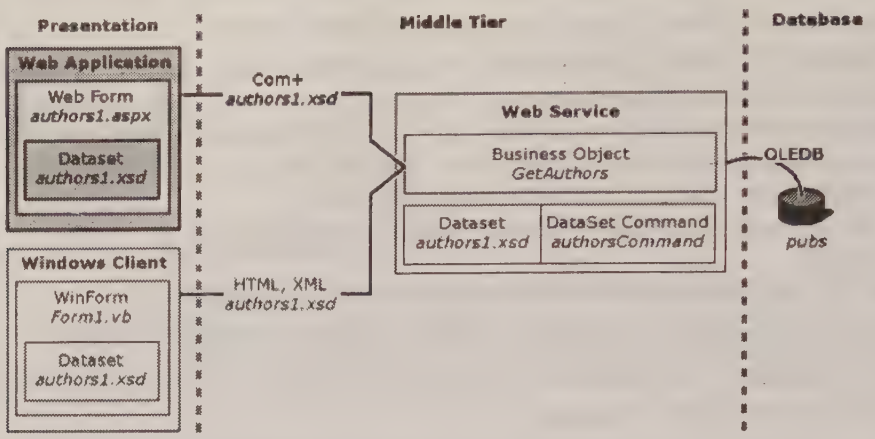
   b. Choose the 'UpdateAuthors Web Service' method from the 'Update' drop-down list to associate it with the update button.
9. Select the 'All Records' option to display all of the records in the authors table as a grid.
10. From the File menu, choose 'Save All'.
11. From the Build menu, choose 'Build'.
12. Press 'F5' to run the application. A window is displayed that contains an empty table with headers from the Authors table in the pubs database.
13. Click 'Load' to populate the table, and then click 'Update' to save changes you make.

In this section, you added a Windows Forms project to your solution to act as a Windows interface. You connected the Windows Form to the Web Service you created in the first section and added Button controls to load and update the data.

21.18 Web User Interface Option

A Web interface helps your application reach a broad variety of client machines and browsers. All of the user-interface processing is accomplished on the Web server rather than the client. Using a Web Forms page, you will create a Web interface that accesses the same middle-tier business object as the Windows interface you created in the previous section.

Web Forms in the distributed application is as follows:



To create a Web Forms page you may follow the following steps:

1. On the File menu, point to Add, and then click 'New Project'.
2. In the 'Add New Project' dialog box, select 'C# Projects' or 'Visual Basic Projects' in the Project Types pane, then click 'Web Application'.
3. Name the project 'AuthorsWebClient'.

A new project is added to Solution Explorer. A Web Forms page called 'WebForm1.aspx' is added to the project and is loaded into the designer.

4. Close the new .aspx file.
5. In Solution Explorer rename the .aspx file to 'AuthorsWebForm.aspx'.
6. Click Yes in the Link Repair window when asked if you want to update links.
7. Add a reference to the Web Service project:
   - In Solution Explorer, right-click the 'AuthorsWinClient' project, and select 'Add Web Reference' from the shortcut menu.
   - Enter the location of the .disco file in your Web Service project (http://servername/AuthorsWebService/AuthorsWebService.disco) and press 'ENTER'.
   - In the 'Components Found' window, select the components you want and click 'Add Reference'.

You can now bind to the DataSet generated in the Web Service.

8. Double-click 'AuthorsWebForm.aspx' in 'Solution Explorer' to open it in the designer.
9. Click the Data tab of the Toolbox, and drag a 'DataFormWizard' onto the form.
10. Choose the Authors1 dataset from the DataSet drop-down list and then click 'Next'.
11. Choose the 'GetAuthors Web Service' method from the drop-down list to associate it with the Load button, then click 'Next'.
12. In the Column window, choose the columns that you want to be shown from the Authors table.

13. Select 'All Records' to display all of the records in the authors table as a grid.
14. From the File menu, choose 'Save All'.
15. From the Build menu, choose 'Build'.
16. Press 'F5' to run the application.

A window is displayed that contains an empty table with headers from the Authors table in the pubs database. Click 'Load' to populate the table and Update to save changes you make.

In this section, you added a Web Application project to your solution to act as a browser interface to your data. You connected the Web Forms page to the Web Service you created in the first section and added code to load the data when the form opens.

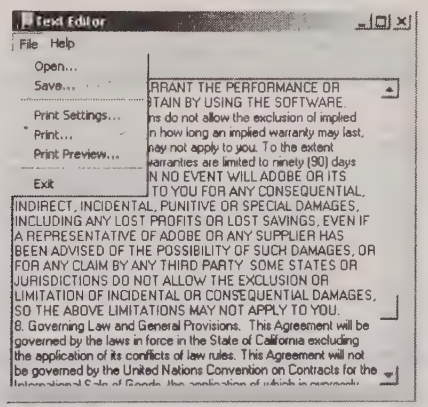
Deploying the Solution: The next step would be to deploy your Web application to a server and to create an installer for your Windows application. For information about deploying your Web application see Walkthrough: Deploying a Web Solution. For information about deploying your Windows application see Walkthrough: Creating an Installer for your Application.

22

MISCELLANEOUS C# OBJECTS

22.1 Printing in C#

It is the use of components that speeds up writing allpications. Printing is carried out through components included in the toolbox. The main component that talks to the printer is the PrintDocument component. To use this component, just call the Print function and intercept the PrintPage event.


Text Editor
File Help
Open...
Save...
Print Settings...
Print...
Print Preview...
Exit
RRANT THE PERFORMANCE OR
TAIN BY USING THE SOFTWARE.
ns do not allow the exclusion of implied
n how long an implied warranty may last,
nay not apply to you. To the extent
warranties are limited to ninety (90) days
N NO EVENT WILL ADOBE OR ITS
TO YOU FOR ANY CONSEQUENTIAL,
INDIRECT, INCIDENTAL, PUNITIVE OR SPECIAL DAMAGES,
INCLUDING ANY LOST PROFITS OR LOST SAVINGS, EVEN IF
A REPRESENTATIVE OF ADOBE OR ANY SUPPLIER HAS
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR
FOR ANY CLAIM BY ANY THIRD PARTY. SOME STATES OR
JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR
LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES,
SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.
8. Governing Law and General Provisions. This Agreement will be
governed by the laws in force in the State of California excluding
the application of its conflicts of law rules. This Agreement will not
be governed by the United Nations Convention on Contracts for the

Following are the routines for this:

Listing 22.1

```
protected void PrintFile_Click(object sender,
System.EventArgs e) {
   printDialog1.Document = ThePrintDocument;
   string strText = this.richTextBox1.Text;
   myReader = new StringReader(strText);
   if (printDialog1.ShowDialog() == DialogResult.OK)
    {
     this.ThePrintDocument.Print();
    }
}
```

```
protected void ThePrintDocument_PrintPage (object sender,
System.Drawing.Printing.PrintPageEventArgs ev) {
   float linesPerPage = 0; float yPosition = 0;
   int count = 0;
   float leftMargin = ev.MarginBounds.Left;
   float topMargin = ev.MarginBounds.Top;
   string line = null;
   Font printFont = this.richTextBox1.Font;
   SolidBrush myBrush = new SolidBrush(Color.Black);
   // Work out the number of lines per page, using the
   //MarginBounds.

   linesPerPage=ev.MarginBounds.Height/printFont.GetHeight(ev.Graphics);
   //Iterate over the string using the StringReader,printing each line.
   while(count < linesPerPage && ((line=myReader.ReadLine()) != null))
   {
         // calculate the next line position based on
         // the height of the font according to the printing device
   yPosition=topMargin+(count*printFont.GetHeight(ev.Graphics));
         //draw the next line in the rich edit control
         ev.Graphics.DrawString(line, printFont, myBrush, leftMargin,
               yPosition, new StringFormat());
         count++;
   }
   // If there are more lines, print another page.
   if  (line != null)
         ev.HasMorePages = true;
   else
         ev.HasMorePages = false;
  myBrush.Dispose();
}
```

Explanation

PrintFile_Click is called from the Print Menu. This routine first assigns the PrintDialog component to the document being printed. The dialog is shown using ShowDialog so that the user can set the print settings and to decide whether to print or not. After ThePrintDocument.Print is called, the printing process begins. The PrintPage event is automatically called by the system to print the desired page information. The PrintPageEventArgs ev contains the Device Contexts ev.Graphics used for drawing to the Printer. In this example we read each line of text from the rich text edit control using a StringReader. The StringReader class gives us the ability to treat a string like a stream and take advantage of the ReadLine function of the StringReader for reading each line out of the text edit control. We can calculate the position of the lines based on margins and the height of the font. It may be noted that, in order to predict the font height for the printer, we pass it the Device context of the printer in the GetHeight function (printFont.GetHeight(ev.Graphics())). This is because the font height on the screen pixels is

different than the font height in printer pixels. The PrintPageEventArg attribute HasMorePages is continually set so that the PrintPage Event is triggered if their are more lines in the textbox than the current page can hold .

22.2 Print Preview

PrintPreview behaves much like the Print seen here:

Listing 22.2

```
protected void PreviewFile_Click (object sender, System.EventArgs  e) {
try {
  string strText = this.richTextBox1.Text;
   // read string from editor window
   myReader = new StringReader(strText);
   PrintPreviewDialog printPreviewDialog1 = new PrintPreviewDialog();
   // instantiate new print preview dialog
   printPreviewDialog1.Document = this.ThePrintDocument ;
   printPreviewDialog1.BorderStyle = FormBorderStyle.Fixed3D ;
   printPreviewDialog1.ShowDialog();
   // Show the print preview dialog,
   //uses print page event to draw preview screen
}
catch(Exception exp)
{
  System.Console.WriteLine(exp.Message.ToString());
}
}
```

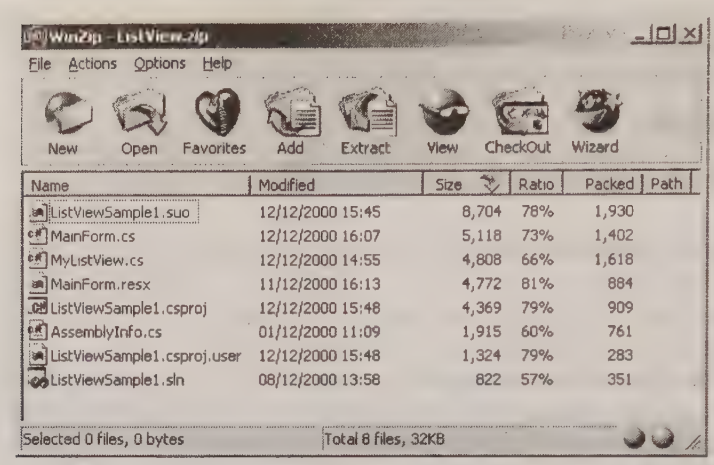
The PrintPreview component also triggers the PrintPage event, but instead of outputting to the printer it outputs to the Print Preview Screen.

22.3 Other Editor Functions

The Open and Save menu items use streams to read and write the files.

22.4 C#.ListView

Sample project that exemplifies sub classing a ListView and adding sorting capabilities.


WinZip - ListView.zip
File Actions Options Help
New Open Favorites Add Extract View CheckOut Wizard
Name Modified Size Ratio Packed Path
ListViewSample1.suo 12/12/2000 15:45 8,704 78% 1,930
MainForm.cs 12/12/2000 16:07 5,118 73% 1,402
MyListView.cs 12/12/2000 14:55 4,808 66% 1,618
MainForm.resx 11/12/2000 16:13 4,772 81% 884
ListViewSample1.csproj 12/12/2000 15:48 4,369 79% 909
AssemblyInfo.cs 01/12/2000 11:09 1,915 60% 761
ListViewSample1.csproj.user 12/12/2000 15:48 1,324 79% 283
ListViewSample1.sln 08/12/2000 13:58 822 57% 351
Selected 0 files, 0 bytes
Total 8 files, 32KB

Listing 22.3

```
// MainForm.cs
namespace ListViewSample1 {
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.WinForms;
/// <summary>
///    MainForm: testing a sub-classed ListView
/// </summary>
public class MainForm : System.WinForms.Form
{
    /// <summary>
    ///    Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components;
    private System.WinForms.Button cmdQuery;
    private System.WinForms.GroupBox groupBox1;
    private System.WinForms.StatusBar sb;
    //
    private System.WinForms.ListView lv;
    private MyListView lv;
    public MainForm()
    {
        //
        // Required for Windows Form Designer support
        //
```

```
        InitializeComponent();
        //
        // Add any constructor code after InitializeComponent call
        //
    }
    /// <summary>
    ///    Clean up any resources being used.
    /// </summary>
    public override void Dispose()
    {
        base.Dispose();
        components.Dispose();
    }
    /// <summary>
    ///  Required method for Designer support - do not modify
    ///  the contents of this method with the code editor.
    /// </summary>
    private void InitializeComponent()
    {
        System.Resources.ResourceManager resources = new
           System.Resources.ResourceManager (typeof(MainForm));
        this.components = new System.ComponentModel.Container ();
        this.lv = new ListViewSample1.MyListView ();
        this.cmdQuery = new System.WinForms.Button ();
        this.groupBox1 = new System.WinForms.GroupBox ();
        this.sb = new System.WinForms.StatusBar ();
        //@this.TrayHeight = 0;
        //@this.TrayLargeIcon = false;
        //@this.SetLocalizable (true);
        //@this.TrayAutoArrange = true;
        lv.Location = (System.Drawing.Point) resources.GetObject
           ("lv.Location");
        lv.Size = (System.Drawing.Size) resources.GetObject
           ("lv.Size");
        lv.Sorting = System.WinForms.SortOrder.Ascending;
        lv.FullRowSelect = true;
        lv.View = System.WinForms.View.Report;
        lv.Dock = (System.WinForms.DockStyle)
           resources.GetObject ("lv.Dock");
        lv.ForeColor = (System.Drawing.Color)
           resources.GetObject ("lv.ForeColor");
        lv.TabIndex = (int) resources.GetObject
           ("lv.TabIndex");
        cmdQuery.Location = (System.Drawing.Point)
           resources.GetObject ("cmdQuery.Location");
```

```
        cmdQuery.Size = (System.Drawing.Size)
            resources.GetObject ("cmdQuery.Size");
        cmdQuery.TabIndex = (int) resources.GetObject
            ("cmdQuery.TabIndex");
        cmdQuery.Anchor = (System.WinForms.AnchorStyles)
            resources.GetObject ("cmdQuery.Anchor");
        cmdQuery.Text = (string) resources.GetObject
            ("cmdQuery.Text");
        cmdQuery.Click += new System.EventHandler
            (this.cmdQuery_Click);
        groupBox1.Dock = (System.WinForms.DockStyle)
            resources.GetObject ("groupBox1.Dock");
        groupBox1.TabIndex = (int) resources.GetObject
            ("groupBox1.TabIndex");
        groupBox1.TabStop = false;
        groupBox1.Size = (System.Drawing.Size)
            resources.GetObject ("groupBox1.Size");
        sb.BackColor = (System.Drawing.Color)
            resources.GetObject ("sb.BackColor");
        sb.Location = (System.Drawing.Point)
            resources.GetObject ("sb.Location");
        sb.Size = (System.Drawing.Size) resources.GetObject
            ("sb.Size");
        sb.TabIndex = (int) resources.GetObject
            ("sb.TabIndex");
        this.Text = (string) resources.GetObject
            ("$this.Text");
        this.AutoScaleBaseSize = new System.Drawing.Size (5,13);
        this.ClientSize = (System.Drawing.Size)
            resources.GetObject ("$this.ClientSize");
        this.Resize += new System.EventHandler
            (this.MainForm_Resize);
        groupBox1.Controls.Add (this.cmdQuery);
        this.Controls.Add (this.groupBox1);
        this.Controls.Add (this.sb);
        this.Controls.Add (this.lv);
    }
    protected void MainForm_Resize (object sender,
            System.EventArgs e)
    {
        lv.AdjustColumnsWidth();
    }
    protected void cmdQuery_Click (object sender,
            System.EventArgs e)
    {
```

```
            PopulateListView();
        }
        private void PopulateListView()
        {
            // Clear Items and Columns
            lv.ClearAll();
            // Add some columns
            lv.AddColumn( "Text Column",
                          MyListView.ColumnTypeEnum.ColumnTypeText,
                          HorizontalAlignment.Left);
         // Alignment is *as allways* ignored in the First column
         lv.AddColumn("NumericColumn",
               MyListView.ColumnTypeEnum.ColumnTypeNumber,
                     HorizontalAlignment.Center);
            lv.AddColumn( "Date Column",
               MyListView.ColumnTypeEnum.ColumnTypeDate,
               HorizontalAlignment.Right );
            // Add some rows
            lv.AddRow( "9",  "9", "12/1/2000");
            lv.AddRow("10", "10", "11/1/2001");
            lv.AddRow("11", "11", "1/1/1998");
            lv.AddRow( "1",  "1", "5/1/1939");
            // Adjust widths...
            lv.AdjustColumnsWidth();
            return;
        }
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        public static void Main(string[] args)
        {
            Application.Run(new MainForm());
        }
    }
}
```

22.5 colorCode

Listing 22.4

```
//MyListView.cs
namespace ListViewSample1 {
using System;
using System.WinForms;
using System.Collections;
```

```
using System.Diagnostics;
/// <summary>
///    MyListView: Subclassing ListView.
///    R. Martinez-Brunet 12/11/2000
/// </summary>
public class MyListView:ListView
{
    private const int LVM_FIRST = 0x1000;
    private const int LVM_SETCOLUMNWIDTH = (LVM_FIRST + 30);
    private const int LVSCW_AUTOSIZE_USEHEADER = -2;
    private Comparer  myComparer;
    private ArrayList colTypes = new ArrayList();
    private class Comparer : IComparer
    {
        public int column = 0;
        public int sign = 1;
        public MyListView.ColumnTypeEnum colType =
           MyListView.ColumnTypeEnum.ColumnTypeText;
        public int Compare( object o1, object o2 )
        {
            ListItem li1 = (ListItem)o1;
            ListItem li2 = (ListItem)o2;
            string s1, s2;
            try
            {
                if ( column == 0)
                {
                    s1 = li1.Text;
                    s2 = li2.Text;
                   return sign*String.Compare(li1.Text,li2.Text);
                }
                else
                {
                    s1 = li1.SubItems[column-1];
                    s2 = li2.SubItems[column-1];
                }
                // Protection against empty strings
                if ( s1.Length == 0 )
                {
                    if ( s2.Length > 0 )
                    {
                        return (-sign);
                    }
                    else
                        return 0;
```

```
                    }
                    else if( s2.Length == 0 )
                    {
                            return sign;
                    }
                    switch( colType )
                    {
                        case ColumnTypeEnum.ColumnTypeNumber:
                            double d1 = double.Parse( s1 );
                            double d2 = double.Parse( s2 );
                            if ( d1 < d2 ) return (-sign);
                            if ( d1 > d2 )
                                return (sign);
                            else
                                return (0);
                        case ColumnTypeEnum.ColumnTypeDate:
                            DateTime t1 = DateTime.Parse( s1 );
                            DateTime t2 = DateTime.Parse ( s2 );
                            return sign*DateTime.Compare(t1,t2);
                        default:
                            return sign*String.Compare(s1, s2);
                    }
                }
                catch
//dunno where this exception is comming from RMB 12/5/2000
                {
                    return 0;
                }
            }
        }
        public enum ColumnTypeEnum
{
   ColumnTypeText = 0,
   ColumnTypeDate = 1,
   ColumnTypeNumber = 2
}
// Cosntructor
        public MyListView()
        {
            // Set the handler for the Column_Click event
            base.ColumnClick += new
          System.WinForms.ColumnClickEventHandler (this.ev_ColumnClick);
            // Set the Comparer for the Sorting
```

```
            myComparer = new Comparer();
            base.ListItemSorter = (IComparer)myComparer;
        }
        public void ClearAll()
        {
            Clear();
            ClearColumns();
        }
        public new void Clear()
        {
            base.ListItems.Clear();
        }
        public void ClearColumns()
        {
            base.Columns.Clear();
            colTypes.Clear();
        }

    public int AddColumn( string caption, ColumnTypeEnum  columnType,
HorizontalAlignment ha )
        {
            ColumnHeader ch = new ColumnHeader();
            ch.Text = caption;
            ch.TextAlign = ha    ;
            base.Columns.Add( ch );
            colTypes.Add( columnType ); // implicit boxing here
            return 0;
        }
        public int AddRow( params string[] values )
        {
            ListItem li = new ListItem(values);
            base.ListItems.Add( li );
            return li.Index;
        }

        public void AdjustColumnWidth(int ColumnIndex)
        {

//- If you include the header in the sizing then the last column will
//- automatically size to fill the remaining listview width.
            /*
   */
//verify that the listview is in report view and that the column exists
            if ( this.View == System.WinForms.View.Report )
            {
                if ( ColumnIndex >= 0 &&  ColumnIndex <
base.Columns.Count )
                {
                    base.SendMessage( LVM_SETCOLUMNWIDTH,
```

```
ColumnIndex, LVSCW_AUTOSIZE_USEHEADER );
                }
            }
        }

        public void AdjustColumnsWidth()
        {
            base.BeginUpdate();
            for(int i = 0; i < base.Columns.Count ; i++)
            {
                AdjustColumnWidth( i );
            }
            base.EndUpdate();
            return;
        }
        public override void Dispose()
        {
            colTypes.Clear();
            base.Dispose();
        }
             protected void ev_ColumnClick (object sender,
System.WinForms.ColumnClickEventArgs e)
        {
            myComparer.column = e.column;
            //Sign and ColType mut be set *before* Sorting is  altered
            myComparer.sign = (base.Sorting ==
SortOrder.Ascending)?-1:1;
            myComparer.colType = (ColumnTypeEnum)
colTypes[e.column];
            base.BeginUpdate();
            switch ( base.Sorting )
            {
                case SortOrder.Ascending:
                    base.Sorting = SortOrder.Descending;
                    break;
                case SortOrder.Descending:
                default:
                    base.Sorting = SortOrder.Ascending;
                    break;
            }
            //AdjustColumnsWidth();
            base.EndUpdate();
        }

    }
}
```

23

Automatic memory management using C#

23.1 Memory Management

Manual memory management requires developers to manage the allocation and de-allocation of blocks of memory.It is both time consuming and tedious. C# provides automatic memory management so that developers are freed from this burdensome task. In the vast majority of cases, this automatic memory management increases code quality and enhances developer productivity without negatively affecting either expressiveness or performance. An example is shown here:

Listing 23.1

```
using System;
public class Stack {
   private Node first = null;
   public bool Empty {
      get {
         return (first == null);
      }
   }
   public object Pop() {
      if (first == null)
         throw new Exception("Cannot Pop from an empty Stack.");
       else {
         object temp = first.Value;
         first = first.Next;
         return temp;
      }
   }
   public void Push(object o) {
```

```
    public void Push(object o) {
       first = new Node(o, first);
    }
    class Node
    {
       public Node Next;
       public object Value;
       public Node(object value): this(value, null) {}
       public Node(object value, Node next) {
          Next = next;
          Value = value;
       }
    }
}
```

This example shows a Stack class implemented as a linked list of Node instances. Node instances are created in the Push method and are garbage collected when no longer needed. A Node instance becomes eligible for garbage collection when it is no longer possible for any code to access it. For instance, when an item is removed from the Stack, the associated Node instance becomes eligible for garbage collection.

Listing 23.2

```
class Test {
   static void Main() {
      Stack s = new Stack();
      for (int i = 0; i < 10; i++)
         s.Push(i);
      while (!s.Empty)
         Console.WriteLine(s.Pop());
   }
}
```

Listing 23.2 shows a test program that uses the Stack class. A Stack is created and initialized with 10 elements, and then assigned the value null. Once the variable 's' is assigned null, the Stack and the associated 10 Node instances become eligible for garbage collection. The garbage collector is permitted to clean up immediately, but is not required to do so.

For developers who are generally content with automatic memory management but sometimes need fine-grained control or that extra iota of performance, C# provides the ability to write 'unsafe' code. Such code can deal directly with pointer types, and fix objects to temporarily prevent the garbage collector from moving them. This 'unsafe' code feature is in fact 'safe' feature from the perspective of both developers and users. Unsafe code must be clearly marked in the code with the modifier unsafe, so developers can't possibly use unsafe features accidently. tally, and the C# compiler and the execution engine work together to ensure that unsafe code cannot masquerade as safe code.

Listing 23.3

```
using System;
class Test {
   unsafe static void WriteLocations(byte[] arr) {
      fixed (byte *p_arr = arr) {
         byte *p_elem = p_arr;
         for (int i = 0; i < arr.Length; i++) {
            byte value = *p_elem;
            string addr = int.Format((int) p_elem, "X");
            Console.WriteLine("arr[{0}] at 0x{1} is {2}", i,
               addr, value);
            p_elem++;
         }
      }
   }
   static void Main() {
      byte[] arr = new byte[] {1, 2, 3, 4, 5};
      WriteLocations(arr);
   }
}
```

Listing 23.3 shows an unsafe method named 'WriteLocations' that fixes an array instance and uses pointer manipulation to iterate over the elements and write out the index, value, and location of each. One possible output of the program is:

```
arr[0] at 0x8E0360 is 1 arr[1] at 0x8E0361 is 2 arr[2] at 0x8E0362 is 3
arr[3] at 0x8E0363 is 4 arr[4] at 0x8E0364 is 5
```

but of course the exact memory locations are subject to change.

23.2 Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework

Implementing proper resource management for your applications can be a difficult, tedious task. Fortunately, in .NET there is: garbage collection (GC). Every program uses resources of one sort or another, memory buffers, screen space, network connections, database resources, and so on. In fact, in an object-oriented environment, every type identifies some resource available for the program's use. Using any of these resources requires that memory to be allocated to represent the type. We will now take the steps required to access a resource which are as follows:

23.3 Allocate memory for the type that represents the resource

1. Initialize the memory to set the initial state of the resource and to make the resource usable.
2. Use the resource by accessing the instance members of the type (repeat as necessary). Tear down the state of the resource to clean up.

3. Free the memory.

This seemingly simple paradigm has been one of the major sources of programming errors. Most often, one forgets to free memory when it is no longer needed or to use memory after it has been freed. These two bugs are worse than most other application bugs because what the consequences will be and when those consequences will occur are typically unpredictable.These two bugs cause resource leaks (memory consumption) and object corruption (destabilization), making your application perform in unpredictable ways at unpredictable times. In fact, there are many tools (such as the Task Manager, the System Monitor ActiveXr Control, CompuWare's BoundsChecker, and Rational's Purify) that are specifically designed to help developers locate these types of bugs.

As you examine GC, you will notice that it completely absolves the developer from tracking memory usage and knowing when to free memory. However, the garbage collector doesn't know anything about the resource represented by the type in memory. This means that a garbage collector can't know how to perform step four ie — tearing down the state of a resource. To get a resource clean up properly, the developer must write code that knows how to properly clean up a resource. In the .NET Framework, the developer writes this code in a Close, Dispose, or Finalize method, whichi will describe later. However, as you will see later, the garbage collector can determine when to call this method automatically. Also, many types represent resources that do not require any cleanup. For example, a Rectangle resource can be completely cleaned up simply by destroying the left, right, width, and height fields maintained in the type's memory. On the other hand, a type that represents a file resource or a network connection resource will require the execution of some explicit clean up code when the resource is to be destroyed. We will see how to accomplish all these later on. For now, let us examine how memory is allocated and how resources are initialized.

23.4 Resource Allocation

The Microsoft .NET common language runtime requires that all resources be allocated from the managed heap. This is similar to a C-runtime heap except that you never free objects from the managed heap objects. They are automatically freed when they are no longer needed by the application. How does the managed heap know when an object is no longer in use by the application? We will address this question shortly. There are several GC algorithms in use today. Each algorithm is fine-tuned for a particular environment in order to provide the best performance.

This section concentrates on the GC algorithm that is used by the common language runtime. Let's start with the basic concepts. When a process is initialized, the runtime reserves a contiguous region of address space that initially has no storage allocated for it. This address space region is the managed heap. The heap also maintains a pointer, which we will call the NextObjPtr. This pointer indicates where the next object is to be allocated within the heap. Initially, the NextObjPtr is set to the base address of the reserved address space region. An application creates an object using the new operator. This operator first makes sure that the bytes required by the new object fit in the reserved region (committing storage if necessary). If the object fits, then NextObjPtr points to the object in the heap, this object's constructor is called, and the new operator returns the address of the object.

```
C:\>mem
arr[0] at 0xB4F670 is 1
arr[1] at 0xB4F671 is 2
arr[2] at 0xB4F672 is 3
arr[3] at 0xB4F673 is 4
arr[4] at 0xB4F674 is 5
```

Managed Heap

At this point, NextObjPtr is incremented past the object so that it points to where the next object will be placed in the heap. The above figure shows a managed heap consisting of three objects: A, B, and C. The next object to be allocated will be placed where 'NextObjPtr' points (immediately after object C).

Now let's look at how the C-runtime heap allocates memory. In a C-runtime heap, allocating memory for an object requires walking though a linked list of data structures. Once a large enough block is found, that block has to be split, and pointers in the linked list nodes must be modified to keep everything intact. For the managed heap, allocating an object simply means adding a value to a pointer—this is very fast by comparison. In fact, allocating an object from the managed heap is nearly as fast as allocating memory from a thread's stack. .

So far, it sounds like the managed heap is far superior to the C- runtime heap due to its speed and simplicity of implementation. Of course, the managed heap gains these advantages because it makes one really big assumption: address space and storage are infinite. This assumption is (without a doubt) ridiculous, and there must be a mechanism employed by the managed heap that allows the heap to make this assumption. This mechanism is called the garbage collector. Let's see how it works. When an application calls the new operator to create an object, there may not be enough address space left in the region to allocate to the object.

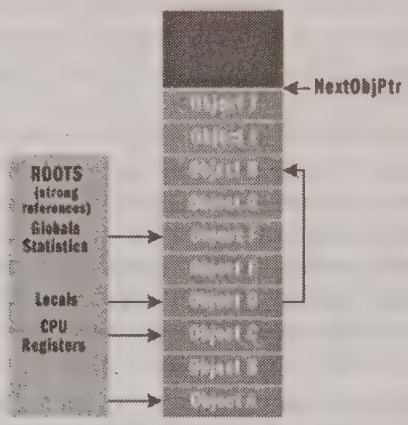
The heap detects this by adding the size of the new object to 'NextObjPtr'. If 'NextObjPtr' is beyond the end of the address space region, then the heap is full and a collection must be performed. In reality, a collection occurs when generation 0 is completely full. Briefly, a generation is a mechanism implemented by the garbage collector in order to improve performance. The idea is that newly created objects are part of a young generation, and objects created early in the application's lifecycle are in an old generation. Separating objects into generations can allow the garbage collector to collect specific generations instead of collecting all objects in the managed heap.

23.5 The Garbage Collection Algorithm

The garbage collector checks to see if there are any objects in the heap that are no longer being used by the application. If such objects exist, then the memory used by these objects can be reclaimed. (If no more memory is available for the heap, then the new operator throws an OutOfMemoryException.) How does the garbage collector know if the application is using an object or not? As you might imagine, this isn't a simple question to answer. Every application has a set of roots. Roots identify storage locations, which refer to objects on the managed heap or to objects that are set to null. For example, all the global and static object pointers in an application are considered part of the application's roots. In addition, any local variable/parameter object pointers on a thread's stack are considered part of the application's roots. Finally, any

CPU registers containing pointers to objects in the managed heap are also considered part of the application's roots.

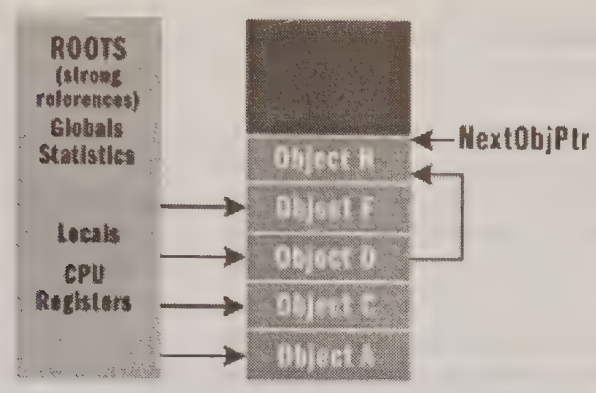
The list of active roots is maintained by the just-in-time (JIT) compiler and common language runtime, and is made accessible to the garbage collector's algorithm. When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage. In other words, it assumes that none of the application's roots refer to any objects in the heap. Now, the garbage collector starts walking the roots and building a graph of all objects reachable from the roots. For example, the garbage collector may locate a global variable that points to an object in the heap. Below figure shows a heap with several allocated objects where the application's roots refer directly to objects A, C, D, and F. All of these objects become part of the graph. When adding object D, the collector notices that this object refers to object H, and object H is also added to the graph. The collector continues to walk through all reachable objects recursively.



Allocated Objects in Heap

Once this part of the graph is complete, the garbage collector checks the next root and walks the objects again. As the garbage collector walks from object to object, if it attempts to add an object to the graph that it previously added, then the garbage collector can stop walking down that path. This serves two purposes.First, it helps performance significantly since it doesn't walk through a set of objects more than once. Second, it prevents infinite loops should you have any circular linked lists of objects.

Once all the roots have been checked, the garbage collector's graph contains the set of all objects that are somehow reachable from the application's roots; any objects that are not in the graph are not accessible by the application, and are therefore considered garbage. The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects (now considered free space). The garbage collector then shifts the non-garbage objects down in memory (using the standard memcpy function that you've known for years), removing all of the gaps in the heap. Of course, moving the objects in memory invalidates all pointers to the objects. So the garbage collector must modify the application's roots so that the pointers point to the objects' new locations. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as ill. Figure in the next page shows the managed heap after a collection.



Managed Heap after Collection

After all the garbage has been identified, all the non-garbage has been compacted, and all the non-garbage pointers have been fixed-up, the NextObjPtr is positioned just after the last non-garbage object. At this point, the new operation is tried again and the resource requested by the application is successfully created. As you can see, a GC generates a significant performance hit, and this is the major downside of using a managed heap. However, keep in mind that GCs only occur when the heap is full and, until then, the managed heap is significantly faster than a C-runtime heap. The runtime's garbage collector also offers some optimizations that greatly improve the performance of garbage collection.

There are a few important things to note at this point. You no longer have to implement any code that manages the lifetime of any resources that your application uses. And notice how the two bugs we discussed at the beginning of this article no longer exist. First, it is not possible to leak resources, since any resource not accessible from your application's roots can be collected at some point. Second, it is not possible to access a resource that is freed, since the resource won't be freed if it is reachable. If it's not reachable, then your application has no way to access it. If garbage collector is so great, you might be wondering why it isn't in ANSI C++. The reason is that a garbage collector must be able to identify an application's roots and must also be able to find all object pointers. The problem with C++ is that it allows casting a pointer from one type to another, and there's no way to know what a pointer refers to. In the common language runtime, the managed heap always knows the actual type of an object, and the metadata information is used to determine which members of an object refer to other objects.

23.6 Finalization

The garbage collector offers an additional feature that you may want to take advantage of: finalization. Finalization allows a resource to gracefully clean up after itself when it is being collected. By using finalization, a resource representing a file or network connection is able to clean itself up properly when the garbage collector decides to free the resource's memory.

Here is an oversimplification of what happens: when the garbage collector detects that an object is garbage, the garbage collector calls the object's Finalize method (if it exists) and then the object's memory is reclaimed. For example, let's say you have the following type (in C#):

Listing 23.4

```
public class BaseObj {
    public BaseObj() {
    }
    protected override void Finalize() {
        // Perform resource cleanup code here...
        // Example: Close file/Close network connection
        Console.WriteLine("In Finalize.");
    }
}
```

Now you can create an instance of this object by calling:

```
BaseObj bo = new BaseObj();
```

Some time in the future, the garbage collector will determine that this object is garbage. When that happens, the garbage collector will see that the type has a Finalize method and will call the method, causing 'In Finalize' to appear in the console window and reclaiming the memory block used by this object.

Many developers who are used to programming in C++ draw an immediate correlation betien a destructor and the Finalize method.However, you must be warned of: object finalization and destructors have very different semantics and it is best to keep aside everything you know about destructors when thinking about finalization. Managed objects never have destructors_period. When designing a type it is best to avoid using a Finalize method. There are several reasons for this:
Finalizable objects get promoted to older generations, which increases memory pressure and prevents the object's memory from being collected when the garbage collector determines the object is garbage. In addition, all objects referred to directly or indirectly by this object get promoted as ill.

23.7 Finalizable objects take longer to allocate

Forcing the garbage collector to execute a Finalize method can significantly affect performance. Remember, each object is finalized. So if we have an array of 10,000 objects, each object must have its Finalize method called.

Finalizable objects may refer to other (non-finalizable) objects, prolonging their lifetime unnecessarily. In fact, you might want to consider breaking a type into two different types: a lightiight type with a Finalize method that doesn't refer to any other objects, and a separate type without a Finalize method that does refer to other objects.You have no control over when the Finalize method will execute. The object may hold on to resources until the next time the garbage collector runs.

When an application terminates, some objects are still reachable and will not have their Finalize method called. This can happen if background threads are using the objects or if objects are created during application shutdown or AppDomain unloading. Additionally, by default, Finalize methods are not called for unreachable objects when an application exits so that the application may terminate quickly. Of course, all operating system resources will be reclaimed, but any

objects in the managed heap are not able to clean up gracefully. You can change this default behavior by calling the System.GC type's RequestFinalizeOnShutdown method. However, you should use this method with care since calling it means that your type is controlling a policy for the entwere application.

The runtime doesn't make any guarantees as to the order in which Finalize methods are called. For example, let's say there is an object that contains a pointer to an inner object. The garbage collector has detected that both objects are garbage. Furthermore, say that the inner object's Finalize method gets called first. Now, the outer object's Finalize method is allowed to access the inner object and call methods on it, but the inner object has been finalized and the results may be unpredictable. For this reason, it is strongly recommended that Finalize methods do not access any inner, member object.

If you determine that your type must implement a Finalize method, then make sure the code executes as quickly as possible. Avoid all actions that would block the Finalize method, including any thread synchronization operation. Also, if you let any exceptions escape the Finalize method, the system just assumes that the Finalize method returned and continues calling other objects' Finalize methods. When the compiler generates code for a constructor, the compiler automatically inserts a call to the base type's constructor. Likewise, when a C++ compiler generates code for a destructor, the compiler automatically inserts a call to the base type's destructor. However, Finalize methods are different from destructors. The compiler has no special knowledge about a Finalize method, so the compiler does not automatically generate code to call a base type's Finalize method. If you want this behavior—and frequently you do—then you must explicitly call the base type's Finalize method from your type's Finalize method:

Listing 23.5

```
public class BaseObj {
    public BaseObj() {
    }
    protected override void Finalize() {
        Console.WriteLine("In Finalize.");
        base.Finalize();    // Call base type's Finalize
    }
}
```

Note that you will usually call the base type's Finalize method as the last statement in the derived type's Finalize method. This keeps the base object alive as long as possible. Since calling a base type Finalize method is common, C# has a syntax that simplifies your work. In C#, the following code

```
class MyObject {

    ~MyObject() {
        ùùù
    }
}
```

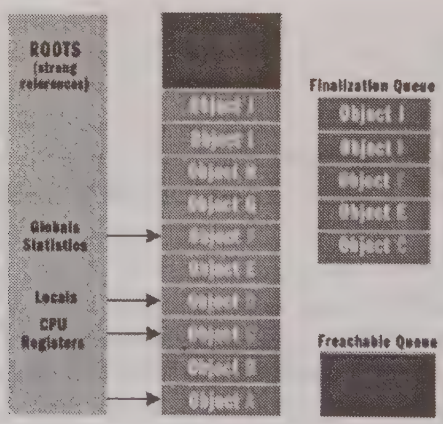
causes the compiler to generate this code:

```
class MyObject {
    protected override void Finalize() {
        ùùù
        base.Finalize();
    }
}
```

Note that this C# syntax looks identical to the C++ language's syntax for defining a destructor. But remember, C# doesn't support destructors. Don't let the identical syntax delude you.

23.8 Finalization Internals

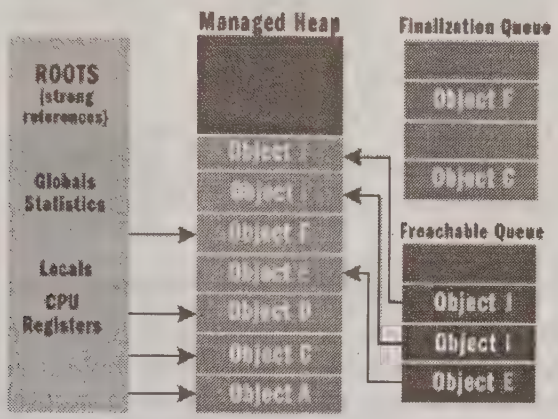
On the surface, finalization seems pretty straightforward: you create an object and when the object is collected, the object's Finalize method is called. But there is more to finalization than this. When an application creates a new object, the new operator allocates the memory from the heap. If the object's type contains a Finalize method, then a pointer to the object is placed on the finalization queue. The finalization queue is an internal data structure controlled by the garbage collector. Each entry in the queue points to an object that should have its Finalize method called before the object's memory can be reclaimed. Below mentined figure shows a heap containing several objects. Some of these objects are reachable from the application's roots, and some are not. When objects C, E, F, I, and J were created, the system detected that these objects had Finalize methods and pointers to these objects were added to the finalization queue.



A Heap with Many Objects

When a GC occurs, objects B, E, G, H, I, and J are determined to be garbage. The garbage collector scans the finalization queue looking for pointers to these objects. When a pointer is found, the pointer is removed from the finalization queue and appended to the freachable queue (pronounced 'F-reachable'). The freachable queue is another internal data structure controlled

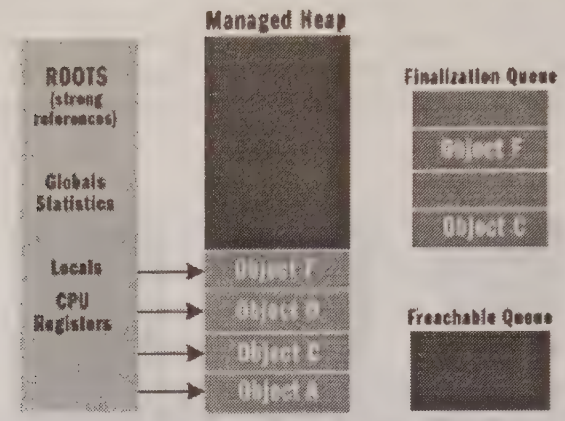
by the garbage collector. Each pointer in the freachable queue identifies an object that is ready to have its Finalize method called. After the collection, the managed heap looks like the Figure below. Here, you see that the memory occupied by objects B, G, and H has been reclaimed because these objects did not have a Finalize method that needed to be called. However, the memory occupied by objects E, I, and J could not be reclaimed because their Finalize method has not been called yet.



Managed Heap after Garbage Collection

There is a special runtime thread dedicated to calling Finalize methods. When the freachable queue is empty (which is usually the case), this thread sleeps. But when entries appear, this thread wakes, removes each entry from the queue, and calls each object's Finalize method. Because of this, you should not execute any code in a Finalize method that makes any assumption about the thread that is executing the code. For example, avoid accessing thread local storage in the Finalize method. The interaction of the finalization queue and the freachable queue is quite fascinating. First, you should know how the freachable queue got its name.

The 'f' is obvious and stands for finalization; every entry in the freachable queue should have its Finalize method called. The 'reachable' part of the name means that the objects are reachable. To put it another way, the freachable queue is considered to be a root just like global and static variables are roots. Therefore, if an object is on the freachable queue, then the object is reachable and is not garbage. In short, when an object is not reachable, the garbage collector considers the object garbage. Then, when the garbage collector moves an object's entry from the finalization queue to the freachable queue, the object is no longer considered garbage and its memory is not reclaimed. At this point, the garbage collector has finished identifying garbage. Some of the objects identified as garbage have been reclassified as not garbage. The garbage collector compacts the reclaimable memory and the special runtime thread empties the freachable queue, executing each object's Finalize method.



Managed Heap after Second Garbage Collection

The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage, since the application's roots don't point to it and the freachable queue no longer points to it. Now the memory for the object is simply reclaimed. The important thing to understand here is that two GCs are required to reclaim memory used by objects that require finalization. In reality, more than two collections may be necessary since the objects could get promoted to an older generation. Above figure shows what the managed heap looks like after the second GC.

23.9 Resurrection

The whole concept of finalization is fascinating. However, there is more to it than what you read so far. You will notice in the previous section that when an application is no longer accessing a live object, the garbage collector considers the object to be dead. However, if the object requires finalization, the object is considered live again until it is actually finalized, and then it is permanently dead. In other words, an object requiring finalization dies, lives, and then dies again. This is an interesting phenomenon called resurrection. Resurrection, as its name implies, allows an object to come back from the dead. We have already described a form of resurrection. When the garbage collector places a reference to the object on the freachable queue, the object is reachable from a root and has come back to life. Eventually, the object's Finalize method is called, no roots point to the object, and the object is dead forever after. But what if an object's Finalize method executed code that placed a pointer to the object in a global or static variable?

Listing 23.6

```
public class BaseObj {
    protected override void Finalize() {
        Application.ObjHolder = this;
    }
}
class Application {
    static public Object ObjHolder;  // Defaults to null ùùù
}
```

In this case, when the object's Finalize method executes, a pointer to the object is placed in a root and the object is reachable from the application's code. This object is now resurrected and the garbage collector will not consider the object to be garbage. The application is free to use the object, but it is very important to note that the object has been finalized and that using the object may cause unpredictable results. Also note: if BaseObj contained members that pointed to other objects (either directly or indirectly), all objects would be resurrected, since they are all reachable from the application's roots. However, be aware that some of these other objects may also have been finalized. In fact, when designing your own object types, objects of your type can get finalized and resurrected totally out of your control. Implement your code so that you handle this carefully. For many types, this means keeping a Boolean flag indicating whether the object has been finalized or not. Then, if methods are called on your finalized object, you might consider throwing an exception. The exact technique to use depends on your type. Now, if some other piece of code sets Application.ObjHolder to null, the object is unreachable. Eventually the garbage collector will consider the object to be garbage and will reclaim the object's storage. Note that the object's Finalize method will not be called because no pointer to the object exists on the finalization queue. There are very few good uses of resurrection, and you really should avoid it if possible. However, when people do use resurrection, they usually want the object to clean itself up gracefully every time the object dies. To make this possible, the GC type offers a method called ReRegisterForFinalize, which takes a single parameter: the pointer to an object.

```
public class BaseObj {
    protected override void Finalize() {
        Application.ObjHolder = this;
        GC.ReRegisterForFinalize(this);
    }
}
```

When this object's Finalize method is called, it resurrects itself by making a root point to the object. The Finalize method then calls ReRegisterForFinalize, which appends the address of the specified object (this) to the end of the finalization queue. When the garbage collector detects that this object is unreachable again, it will queue the object's pointer on the freachable queue and the Finalize method will get called again. This specific example shows how to create an object that constantly resurrects itself and never dies, which is usually not desirable. It is far more common to conditionally set a root to reference the object inside the Finalize method. Make sure that you call ReRegisterForFinalize no more than once per resurrection, or the object will have its Finalize method called multiple times. This happens because each call to ReRegisterForFinalize appends a new entry to the end of the finalization queue. When an object is determined to be garbage, all of these entries move from the finalization queue to the freachable queue, calling the object's Finalize method multiple times.

23.10 Forcing an Object to Clean Up

If you can, you should try to define objects that do not require any clean up. Unfortunately, for many objects, this is simply not possible. So for these objects, you must implement a Finalize method as part of the type's definition. However, it is also recommended that you add an additional method to the type that allows a user of the type to explicitly clean up the object when they want. By convention, this method should be called Close or Dispose.

In general, you use Close if the object can be reopened or reused after it has been closed. You also use Close if the object is generally considered to be closed, such as a file. On the other hand, you would use Dispose if the object should no longer be used at all after it has been disposed. For example, to delete a System.Drawing.Brush object, you call its Dispose method. Once disposed, the Brush object cannot be used, and calling methods to manipulate the object may cause exceptions to be thrown. If you need to work with another Brush, you must construct a new Brush object. Now, let's look at what the Close/Dispose method is supposed to do.

The System.IO.FileStream type allows the user to open a file for reading and writing. To improve performance, the type's implementation makes use of a memory buffer. Only when the buffer fills does the type flush the contents of the buffer to the file. Let's say that you create a new FileStream object and write just a few bytes of information to it. If these bytes don't fill the buffer, then the buffer is not written to disk. The FileStream type does implement a Finalize method, and when the FileStream object is collected the Finalize method flushes any remaining data from memory to disk and then closes the file. But this approach may not be good enough for the user of the FileStream type. Let's say that the first FileStream object has not been collected yet, but the application wants to create a new FileStream object using the same disk file. In this scenario, the second FileStream object will fail to open the file if the first FileStream object had the file open for exclusive access. The user of the FileStream object must have some way to force the final memory flush to disk and to close the file. If you examine the FileStream type's documentation, you will see that it has a method called Close. When called, this method flushes the remaining data in memory to the disk and closes the file. Now the user of a FileStream object has control of the object's behavior. But another problem arises now: what should the FileStream's Finalize method do when the FileStream object is collected? Obviously, the answer is nothing. In fact, there is no reason for the FileStream's Finalize method to execute at all if the application has explicitly called the Close method. You know that Finalize methods are discouraged, and in this scenario you are going to have the system call a Finalize method that should do nothing. It seems like there ought to be a way to suppress the system's calling of the object's Finalize method. Fortunately, there is. The System.GC type contains a static method, SuppressFinalize, that takes a single parameter, the address of an object.

When you call SuppressFinalize, it turns on a bit flag associated with the object. When this flag is on, the runtime knows not to move this object's pointer to the freachable queue, preventing the object's Finalize method from being called. Let's examine another related issue. It is very common to use a StreamWriter object with a FileStream object.

Listing 23.7

```
FileStream fs = new FileStream("C:\\SomeFile.txt",
   FileMode.Open, FileAccess.Write, FileShare.Read);
StreamWriter sw = new StreamWriter(fs); sw.Write ("Hi there");
// The call to Close below is what you should do
sw.Close();
// NOTE: StreamWriter.Close closes the FileStream.The FileStream
//should not be explicitly closed in this scenario
```

Notice that the StreamWriter's constructor takes a FileStream object as a parameter. Internally, the StreamWriter object saves the FileStream's pointer. Both of these objects have internal data buffers that should be flushed to the file when you are finished accessing the file. Calling

the StreamWriter's Close method writes the final data to the FileStream and internally calls the FileStream's Close method, which writes the final data to the disk file and closes the file. Since StreamWriter's Close method closes the FileStream object associated with it, you should not call 'fs'.Close yourself. What do you think would happen if you removed the two calls to Close? Well, the garbage collector would correctly detect that the objects are garbage and the objects would get finalized. But, the garbage collector doesn't guarantee the order in which the Finalize methods are called. So if the FileStream gets finalized first, it closes the file. Then when the StreamWriter gets finalized, it would attempt to write data to the closed file, raising an exception. Of course, if the StreamWriter got finalized first, then the data would be safely written to the file. How did Microsoft solve this problem? Making the garbage collector finalize objects in a specific order is impossible because objects could contain pointers to each other and there is no way for the garbage collector to correctly guess the order to finalize these objects. So, here is Microsoft's solution: the ' StreamWriter' type doesn't implement a Finalize method at all. Of course, this means that forgetting to explicitly close the StreamWriter object guarantees data loss. Microsoft expects that developers will see this consistent loss of data and will fix the code by inserting an explicit call to Close. As stated earlier, the 'SuppressFinalize' method simply sets a bit flag indicating that the object's Finalize method should not be called. However, this flag is reset when the runtime determines that it's time to call a Finalize method. This means that calls to ReRegisterForFinalize cannot be balanced by calls to SuppressFinalize. ReRegisterForFinalize and SuppressFinalize are implemented the way they are for performance reasons. As long as each call to SuppressFinalize has an intervening call to ReRegisterForFinalize, everything works. It is up to you to ensure that you do not call ReRegisterForFinalize or SuppressFinalize multiple times consecutively, or multiple calls to an object's Finalize method can occur.

Weak References

When a root points to an object, the object cannot be collected because the application's code can reach the object. When a root points to an object, it is called a strong reference to the object. However, the garbage collector also supports weak references. Weak references allow the garbage collector to collect the object, but they also allow the application to access the object. How can this be? It all comes down to timing. If only weak references to an object exist and the garbage collector runs, the object is collected and when the application later attempts to access the object, the access will fail. On the other hand, to access a weakly referenced object, the application must obtain a strong reference to the object.

If the application obtains this strong reference before the garbage collector collects the object, then the garbage collector can't collect the object because a strong reference to the object exists. Why might you use weak references? Well, there are some data structures that are created easily, but require a lot of memory. For example, you might have an application that needs to know all the directories and files on the user's hard drive. You can easily build a tree that reflects this information and as your application runs, you will refer to the tree in memory instead of actually accessing the user's hard disk.

This procedure greatly improves the performance of your application. The problem is that the tree could be extremely large, requiring quite a bit of memory. If the user starts accessing a different part of your application, the tree may no longer be necessary and is wasting valuable memory. You could delete the tree, but if the user switches back to the first part of your

application, you will need to reconstruct the tree again. Weak references allow you to handle this scenario quite easily and efficiently. When the user switches away from the first part of the application, you can create a weak reference to the tree and destroy all strong references. If the memory load is low for the other part of the application, then the garbage collector will not reclaim the tree's objects. When the user switches back to the first part of the application, the application attempts to obtain a strong reference for the tree. If successful, the application doesn't have to traverse the user's hard drive again. The WeakReference type offers two constructors:

WeakReference(Object target); WeakReference(Object target, Boolean trackResurrection)
The target parameter identifies the object that the WeakReference object should track. The trackResurrection parameter indicates whether the WeakReference object should track the object after it has had its Finalize method called. Usually, false is passed for the trackResurrection parameter and the first constructor creates a WeakReference that does not track resurrection. For convenience, a weak reference that does not track resurrection is called a short weak reference, while a weak reference that does track resurrection is called a long weak reference. If an object's type doesn't offer a Finalize method, then short and long weak references behave identically.

It is strongly recommended that you avoid using long weak references. Long weak references allow you to resurrect an object after it has been finalized and the state of the object is unpredictable. Once you've created a weak reference to an object, you usually set the strong reference to the object to null.If any strong reference remains, the garbage collector will be unable to collect the object. To use the object again, you must turn the weak reference into a strong reference. You accomplish this simply by calling the WeakReference object's Target property and assigning the result to one of your application's roots. If the Target property returns null, then the object was collected. If the property does not return null, then the root is a strong reference to the object and the code may manipulate the object. As long as the strong reference exists, the object cannot be collected.

Weak Reference Internals From the previous discussion, it should be obvious that WeakReference objects do not behave like other object types. Normally, if your application has a root that refers to an object and that object refers to another object, then both objects are reachable and the garbage collector cannot reclaim the memory in use by either object. However, if your application has a root that refers to a WeakReference object, then the object referred to by the WeakReference object is not considered reachable and may be collected.

To fully understand how weak references work, let's look inside the managed heap again. The managed heap contains two internal data structures whose sole purpose is to manage weak references: the short weak reference table and the long weak reference table. These two tables simply contain pointers to objects allocated within the managed heap. Initially, both tables are empty. When you create a WeakReference object, an object is not allocated from the managed heap. Instead, an empty slot in one of the weak reference tables is located; short weak references use the short weak reference table and long weak references use the long weak reference table. Once an empty slot is found, the value in the slot is set to the address of the object you wish to track the object's pointer is passed to the WeakReference's constructor. The value returned from the new operator is the address of the slot in the WeakReference table. Obviously, the two

weak reference tables are not considered part of an application's roots or the garbage collector would not be able to reclaim the objects pointed to by the tables. Now, here's what happens when a garbage collection (GC) runs:

As we know, the garbage collector builds a graph of all the reachable objects.The garbage collector scans the short weak reference table. If a pointer in the table refers to an object that is not part of the graph, then the pointer identifies an unreachable object and the slot in the short weak reference table is set to null.

The garbage collector scans the finalization queue. If a pointer in the queue refers to an object that is not part of the graph, then the pointer identifies an unreachable object and the pointer is moved from the finalization queue to the freachable queue. At this point, the object is added to the graph since the object is now considered reachable.

The garbage collector scans the long weak reference table. If a pointer in the table refers to an object that is not part of the graph (which now contains the objects pointed to by entries in the freachable queue), then the pointer identifies an unreachable object and the slot is set to null.

The garbage collector compacts the memory, squeezing out the holes left by the unreachable objects.

Once you understand the logic of the garbage collection process, it is easy to understand how weak references work. Accessing the WeakReference's Target property causes the system to return the value in the appropriate weak reference table's slot. If null is in the slot, the object was collected. A short weak reference doesn't track resurrection. This means that the garbage collector sets the pointer to null in the short weak reference table as soon as it has determined that the object is unreachable. If the object has a Finalize method, the method has not been called yet so the object still exists. If the application accesses the WeakReference object's Target property, then null will be returned even though the object actually still exists. A long weak reference tracks resurrection. This means that the garbage collector sets the pointer to null in the long weak reference table when the object's storage is reclaimable. If the object has a Finalize method, the Finalize method has been called and the object was not resurrected.

Generations:

One feature of the garbage collector that exists purely to improve performance is called generations. A generational garbage collector (also known as an ephemeral garbage collector) makes the following assumptions:

The newer an object is, the shorter its lifetime will be.

The older an object is, the longer its lifetime will be.

Newer objects tend to have strong relationships to each other and are frequently accessed around the same time.

Compacting a portion of the heap is faster than compacting the whole heap.

Of course, many studies have demonstrated that these assumptions are valid for a very large set of existing applications. So, let's discuss how these assumptions have influenced the implementation of the garbage collector. When initialized, the managed heap contains no objects. Objects added to the heap are said to be in generation 0. Stated simply, objects in generation 0 are young objects that have never been examined by the garbage collector.

Now, if more objects are added to the heap, the heap fills and a garbage collection must occur. When the garbage collector analyzes the heap, it builds the graph of garbage (shown here in purple) and non-garbage objects. Any objects that survive the collection are compacted into the left-most portion of the heap. These objects have survived a collection, are older, and are now considered to be in generation 1.

As even more objects are added to the heap, these new, young objects are placed in generation 0. If generation 0 fills again, a GC is performed. This time, all objects in generation 1 that survive are compacted and considered to be in generation 2. All survivors in generation 0 are now compacted and considered to be in generation 1. Generation 0 currently contains no objects, but all new objects will go into generation 0.

Generations 0, 1, and 2Currently, generation 2 is the highest generation supported by the runtime's garbage collector. When future collections occur, any surviving objects currently in generation 2 simply stay in generation 2.

Generational GC Performance Optimizations as stated earlier, improves performance. When the heap fills and a collection occurs, the garbage collector can choose to examine only the objects in generation 0 and ignore the objects in any greater generations. After all, the newer an object is, the shorter its lifetime is expected to be. So, collecting and compacting generation 0 objects is likely to reclaim a significant amount of space from the heap and be faster than if the collector had examined the objects in all generations. This is the simplest optimization that can be obtained from generational GC.

A generational collector can offer more optimizations by not traversing every object in the managed heap. If a root or object refers to an object in an old generation, the garbage collector can ignore any of the older objects' inner references, decreasing the time required to build the graph of reachable objects. Of course, it is possible that an old object refers to a new object. So that these objects are examined, the collector can take advantage of the system's write-watch support (provided by the Win32r GetWriteWatch function in Kernel32.dll). This support lets the collector know which old objects (if any) have been written to since the last collection. These specific old objects can have their references checked to see if they refer to any new objects. If collecting generation 0 doesn't provide the necessary amount of storage, then the collector can attempt to collect the objects from generations 1 and 0. If all else fails, then the collector can collect the objects from all generations—2, 1, and 0. The exact algorithm used by the collector to determine which generations to collect is one of those areas that Microsoft will be tweaking forever.

Most heaps (like the C runtime heap) allocate objects wherever they find free space. Therefore, if we create several objects consecutively, it is quite possible that these objects will be separated by megabytes of address space. However, in the managed heap, allocating several objects consecutively ensures that the objects are contiguous in memory. One of the assumptions stated earlier was that newer objects tend to have strong relationships to each other and are frequently accessed around the same time. Since new objects are allocated contiguously in memory, you gain performance from locality of reference. More specifically, it is highly likely that all the objects can reside in the CPU's cache. Your application will access these objects with phenomenal speed since the CPU will be able to perform most of its manipulations without having cache misses which forces RAM access. Microsoft's performance tests show that managed heap

allocations are faster than standard allocations performed by the Win32 HeapAlloc function. These tests also show that it takes less than 1 millisecond on a 200Mhz Pentium to perform a full GC of generation 0. It is Microsoft's goal to make GCs take no more time than an ordinary page fault.

Direct Control with System.GC The System.GC type allows your application some direct control over the garbage collector. For starters, you can query the maximum generation supported by the managed heap by reading the GC.MaxGeneration property. Currently, the GC.MaxGeneration property always returns 2. It is also possible to force the garbage collector to perform a collection by calling one of the two methods shown here:

```
void GC.Collect(Int32 Generation)
void GC.Collect()
```

The first method allows you to specify which generation to collect. You may pass any integer from 0 to GC.MaxGeneration, inclusive. Passing 0 causes generation 0 to be collected; passing 1 causes generation 1 and 0 to be collected; and passing 2 causes generation 2, 1, and 0 to be collected. The version of the Collect method that takes no parameters forces a full collection of all generations and is equivalent to calling:

```
GC.Collect(GC.MaxGeneration);
```

Under most circumstances, you should avoid calling any of the Collect methods; it is best to just let the garbage collector run on its own accord. However, since your application knows more about its behavior than the runtime does, you could help matters by explicitly forcing some collections. For example, it might make sense for your application to force a full collection of all generations after the user saves his data file. Imagine that Internet browsers performing a full collection when pages are unloaded. You might also want to force a collection when your application is performing other lengthy operations; this hides the fact that the collection is taking processing time and prevents a collection from occurring when the user is interacting with your application.The GC type also offers a WaitForPendingFinalizers method. This method simply suspends the calling thread until the thread processing the freachable queue has emptied the queue, calling each object's Finalize method. In most applications, it is unlikely that you will ever have to call this method. Lastly, the garbage collector offers two methods that allow you to determine which generation an object is currently in:
Int32 GetGeneration(Object obj) Int32 GetGeneration(WeakReference wr)

The first version of GetGeneration takes an object reference as a parameter, and the second version takes a WeakReference reference as a parameter. Of course, the value returned will be somewhere betien 0 and GC.MaxGeneration, inclusive. The code in Fig. 5 will help you understand how generations work. It also demonstrates the use of the garbage collection methods just discussed.

Performance for Multithreaded Applications: In the previous section, we explained the GC algorithm and optimizations. However, there was a big assumption made during that discussion: only one thread is running. In the real world, it is quite likely that multiple threads will be accessing the managed heap or at least manipulating objects allocated within the managed heap. When one thread sparks a collection, other threads must not access any objects (including object references on its own stack) since the collector is likely to move these objects, changing their memory locations. So, when the garbage collector wants to start a collection, all threads

executing managed code must be suspended. The runtime has a few different mechanisms that it uses to safely suspend threads so that a collection may be done. The reason there are multiple mechanisms is to keep threads running as long as possible and to reduce overhead as much as possible.We will not go into all the details here, but suffice it to say that Microsoft has done a lot of work to reduce the overhead involved with performing a collection. Microsoft will continue to modify these mechanisms over time to help ensure efficient garbage collections. The following paragraphs describe a few of the mechanisms that the garbage collector employs when applications have multiple threads: Fully Interruptible Code When a collection starts, the collector suspends all application threads. The collector then determines where a thread got suspended and using tables produced by the just-in-time (JIT) compiler, the collector can tell where in a method the thread stopped, what object references the code is currently accessing, and where those references are held (in a variable, CPU register, and so on). Hijacking the collector can modify a thread's stack so that the return address points to a special function. When the currently executing method returns, this special function will execute, suspending the thread. Stealing the thread's execution path this way is referred to as hijacking the thread. When the collection is complete, the thread will resume and return to the method that originally called it.

Safe Points: As the JIT compiler compiles a method, it can insert calls to a special function that checks if a GC is pending. If so, the thread is suspended, the GC runs to completion, and the thread is then resumed. The position where the compiler inserts these method calls is called a GC safe point. Note that thread hijacking allows threads that are executing unmanaged code to continue execution while a garbage collection is occurring. This is not a problem since unmanaged code is not accessing objects on the managed heap unless the objects are pinned and don't contain object references. A pinned object is one that the garbage collector is not allowed to move in memory.

If a thread that is currently executing unmanaged code returns to managed code, the thread is hijacked and is suspended until the GC completes. In addition to the mechanisms we just mentioned, the garbage collector offers some additional improvements that enhance the performance of object allocations and collections when applications have multiple threads. Synchronization-free Allocations On a multiprocessor system, generation 0 of the managed heap is split into multiple memory arenas using one arena per thread. This allows multiple threads to make allocations simultaneously so that exclusive access to the heap is not required. Scalable Collections On a multiprocessor system running the server version of the execution engine (MSCorSvr.dll), the managed heap is split into several sections, one per CPU. When a collection is initiated, the collector has one thread per CPU; all threads collect their own sections simultaneously. The workstation version of the execution engine (MSCorWks.dll) doesn't support this feature.

BOOK PARADISE
33993/18B
8177221426
WD/1D8-1120
12/09/2001
133(2)
RS 299.00

The Complete Guide to

C# Programming

This book deals exhaustively with C# (C Sharp) Programming. If you are a programmer, working on Windows with C, C++ or VB, you can easily grasp the nuances of C# Programming. Starting with an introduction and overview on C#, the book covers every aspect of the C# language. It also includes XML, ASP.NET Framework, Microsoft's .NET Framework & Platform, Exceptions and Error Handling in C#. A separate chapter elaborates the future implementations of C# on the various operating platforms.

This book will be of immense help to those budding programmers and developers who wish to procure an all-round knowledge of C#.

Lastly, the book provides detailed description on projects in C#, differences between C++ and C# and interfacing with COM (Component Object Model).

CATEGORY:
PROGRAMMING/C#

READER LEVEL:
INTERMEDIATE TO ADVANCED

PRICE: Rs. 299/-

Features of this Book

- Concise introduction to all the essential features of the C# language
- Facilitates excellent in-depth learning for Beginners, as well as, Experts.
- Heavily coded for better assimilation of concepts.
- Details on the .NET Framework and Platform and explains how C# fits into it.
- Provides a vivid comparison of C#, C++ & VB.NET.
- Explains how to access databases with ADO.NET.
- Arms the concepts with a plethora of examples.
- Discusses how to create and deploy .NET assemblies.
- Equips programmers to develop Object Oriented Programmes in C#.
- Dwells on real-life situations.

ISBN: 81-7722-142-6

Published by

dreamtech Press

Sold by

Hungry Minds™

IDG Books India (P) Ltd.

4435/7 Ansari Road, Daryaganj,
New Delhi-110002
Ph. 91-11-3260877, 3270877
Fax: 91-11-3275895
E-mail: idgbooks@vsnl.com

www.idgbooksindia.com